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, **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

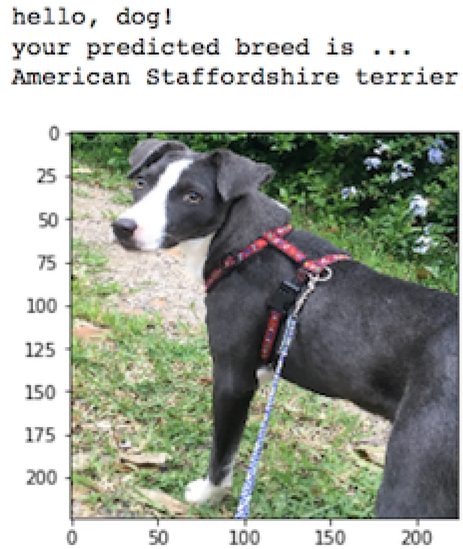
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes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

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

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

Why We're Here

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



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

The Road Ahead

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

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

Step 0: Import Datasets

Import Dog Dataset

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

- `train_files`, `valid_files`, `test_files` - numpy arrays containing file paths to images
- `train_targets`, `valid_targets`, `test_targets` - numpy arrays containing onehot-encoded classification labels

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
iles, 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.
```

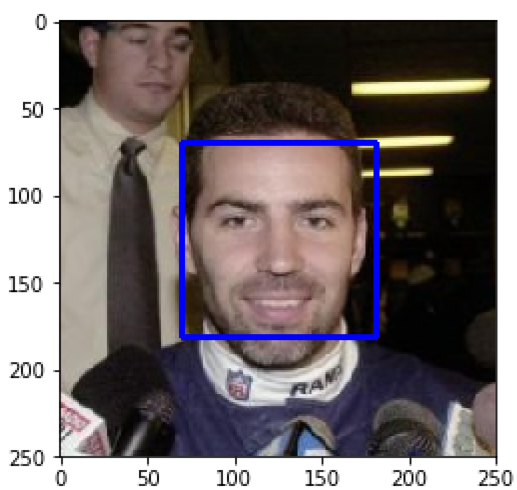
Step 1: Detect Humans

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_frontalface_al
t.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:99% of human are correct and 11% dog files are found as human, which is not correct. That means Haar cascades detects dogs as human. We need tweak in such a way that number of dogs detected should be minimal (probably less than 2%) with the Haar cascades. Also, Haar cascades is not suitable for human face detection, it only detects as front face and not takes care of side face (some angle of rotation).

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.

human_human = 0
for human in human_files_short:
   human_human += face_detector(human)
print('Number of human detected {} in human files {}, percentage is {:.2%}.'\
     .format(human_human, len(human_files_short),(human_human/len(human_files_
short))))

human_dogs = 0
for dog in dog_files_short:
   human_dogs += face_detector(dog)
print('Number of human detected {} in dog files {}, percentage {:.2%}.'\
     .format(human_dogs, len(dog_files_short),(human_dogs/len(dog_files_short
))))
```

```
Number of human detected 99 in human files 100, percentage is 99.00%.
Number of human detected 11 in dog files 100, percentage 11.00%.
```

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: Added some control parameters and got less percentage for human and found 2% as dogs detected in human.

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on each of the datasets.

In [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.
def face_detector_another(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.2,
    minNeighbors=5,
    minSize=(30, 30),
    flags=cv2.CASCADE_SCALE_IMAGE)
    return len(faces) > 0

human_human_another = 0
for human in human_files_short:
   human_human_another += face_detector_another(human)
print('Number of human detected {} in human files {}, percentage is {:.2%}
.'\
     .format(human_human_another, len(human_files_short),(human_human_anoth
er/len(human_files_short) )))

human_dogs_another = 0
for dog in dog_files_short:
   human_dogs_another += face_detector_another(dog)
print('Number of human detected {} in dog files {}, percentage {:.2%}.'\
     .format(human_dogs_another, len(dog_files_short),(human_dogs_another/l
en(dog_files_short) )))
```

```
Number of human detected 94 in human files 100, percentage is 94.00%.
Number of human detected 2 in dog files 100, percentage 2.00%.
```

Step 2: Detect Dogs

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

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

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

Pre-process the Data

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

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

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

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

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

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

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

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

In [8]:

```
from keras.preprocessing import image
from tqdm import tqdm

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

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

Making Predictions with ResNet-50

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

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

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

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

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

Write a Dog Detector

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

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

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

(IMPLEMENTATION) Assess the Dog Detector

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

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

Answer: Improved a lot, 1% dog detected in human files and where 100% of dogs detected.

In [11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
dogs_human = 0
for human in human_files_short:
   dogs_human += dog_detector(human)
print('Number of dogs detected {} in human files {}, percentage is {:.2%}.'\
     .format(dogs_human, len(human_files_short),(dogs_human/len(human_files_sh
ort) )))

dogs_dogs = 0
for dog in dog_files_short:
   dogs_dogs += dog_detector(dog)
print('Number of dogs detected {} in dog files {}, percentage {:.2%}.'\
     .format(dogs_dogs, len(dog_files_short),(dogs_dogs/len(dog_files_short)
)))
```

```
Number of dogs detected 1 in human files 100, percentage is 1.00%.
Number of dogs detected 100 in dog files 100, percentage 100.00%.
```

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

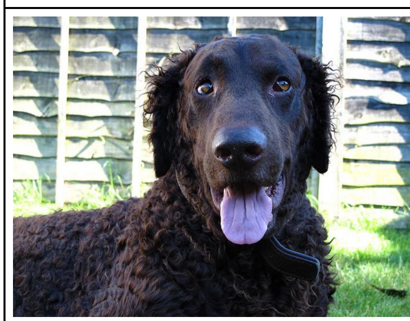
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

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

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

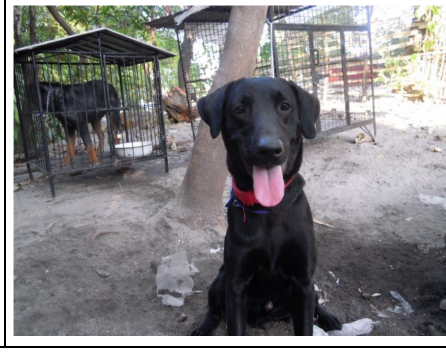
Brittany	Welsh Springer Spaniel
	

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

Curly-Coated Retriever	American Water Spaniel
	

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

Yellow Labrador	Chocolate Labrador	Black Labrador

Yellow Labrador	Chocolate Labrador	Black Labrador
		

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

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

Pre-process the Data

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

In [12]:
```
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
#print('Train tensors shape:', train_tensors.shape)
#print('Train tensors:', train_tensors.shape[0])
#print('Valid tensors:', valid_tensors.shape[0])
#print('Test tensors:', test_tensors.shape[0])
```

```
100%|██████████| 6680/6680 [01:02<00:00, 106.27it/s]
100%|██████████| 835/835 [00:07<00:00, 114.16it/s]
100%|██████████| 836/836 [00:07<00:00, 118.82it/s]
```

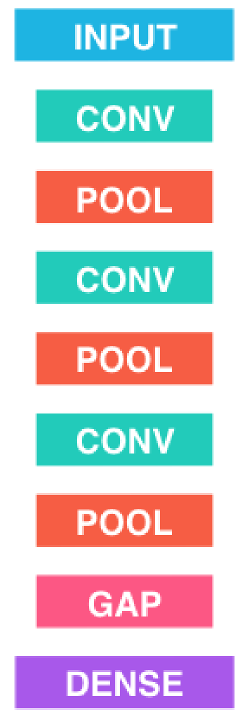
(IMPLEMENTATION) Model Architecture

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

```
model.summary()
```

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

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



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

Answer: The CNN developed based on the CNN summary. The CNN can be changed and got test accuracy of 6% for 20 epochs, if epochs value is 50 or 100, the accuracy is improved. CNN architecture start with filters 16 and then go 32, 64, etc., fixed the kernel as 2 (height and width), made padding as "valid" means "no padding" since we don't need to pad the input such that the output has the same length as the original input. The architecture works good since the input shapes matches with the first conv 2d but test accuracy is not that good.

In [13]:

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

model = Sequential()

### TODO: Define your architecture.
#model.add(Conv2D(16, kernel_size=2, padding='same', activation='relu', inp
ut_shape=(32,32,3)))
model.add(Conv2D(16, kernel_size=2, padding='valid', activation='relu',
                 input_shape=train_tensors.shape[1:]))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32,kernel_size=2, padding='valid', activation='rel
u'))
model.add(MaxPooling2D(pool_size=2))
#model.add(Dropout(0.3))
model.add(Conv2D(filters=64,kernel_size=2, padding='valid', activation='rel
u'))
model.add(MaxPooling2D(pool_size=2))
#model.add(Dropout(0.25))
model.add(GlobalAveragePooling2D())
model.add(Dense(133, 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 [14]:

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

(IMPLEMENTATION) Train the Model

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

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

In [15]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 20

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

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

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8835 - acc: 0.
0098Epoch 00000: val_loss improved from inf to 4.86562, saving model to saved
_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8835 - acc: 0.0097
- val_loss: 4.8656 - val_acc: 0.0108
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8585 - acc: 0.
0147Epoch 00001: val_loss improved from 4.86562 to 4.83565, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8586 - acc: 0.0147
- val_loss: 4.8357 - val_acc: 0.0192
Epoch 3/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8125 - acc: 0.
0167Epoch 00002: val_loss improved from 4.83565 to 4.79871, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8124 - acc: 0.0166
- val_loss: 4.7987 - val_acc: 0.0204
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7695 - acc: 0.
0183Epoch 00003: val_loss improved from 4.79871 to 4.76858, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7695 - acc: 0.0183
- val_loss: 4.7686 - val_acc: 0.0240
Epoch 5/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7348 - acc: 0.
0216Epoch 00004: val_loss improved from 4.76858 to 4.75664, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7346 - acc: 0.0217
- val_loss: 4.7566 - val_acc: 0.0228
Epoch 6/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7045 - acc: 0.
0272Epoch 00005: val_loss improved from 4.75664 to 4.73780, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7051 - acc: 0.0272
- val_loss: 4.7378 - val_acc: 0.0216
Epoch 7/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6785 - acc: 0.
0296- ETA: 1s -Epoch 00006: val_loss improved from 4.73780 to 4.71348, saving
model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6788 - acc: 0.0296
- val_loss: 4.7135 - val_acc: 0.0204
Epoch 8/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6616 - acc: 0.
0330- ETA: 4s - lo - ETA: 2s - loss: 4.6582 - acc:  - ETA: Epoch 00007: val_l
oss did not improve
6680/6680 [==============================] - 11s - loss: 4.6612 - acc: 0.0331
- val_loss: 4.7384 - val_acc: 0.0287
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6434 - acc: 0.
0347Epoch 00008: val_loss improved from 4.71348 to 4.68926, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 11s - loss: 4.6438 - acc: 0.0346
- val_loss: 4.6893 - val_acc: 0.0287
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6271 - acc: 0.
```

```
0365- ETA: 5s - loss: 4 - ETA: 1sEpoch 00009: val_loss improved from 4.68926
to 4.67146, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6268 - acc: 0.0365
- val_loss: 4.6715 - val_acc: 0.0275
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6086 - acc: 0.
0399Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 11s - loss: 4.6088 - acc: 0.0398
- val_loss: 4.6865 - val_acc: 0.0311
Epoch 12/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.5913 - acc: 0.
0416Epoch 00011: val_loss improved from 4.67146 to 4.65436, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.5922 - acc: 0.0415
- val_loss: 4.6544 - val_acc: 0.0323
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.5752 - acc: 0.
0416Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.5752 - acc: 0.0415
- val_loss: 4.6577 - val_acc: 0.0359
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.5577 - acc: 0.
0429Epoch 00013: val_loss improved from 4.65436 to 4.63202, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.5573 - acc: 0.0428
- val_loss: 4.6320 - val_acc: 0.0323
Epoch 15/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.5369 - acc: 0.
0464- - ETA: 1s - losEpoch 00014: val_loss improved from 4.63202 to 4.61243,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.5380 - acc: 0.0463
- val_loss: 4.6124 - val_acc: 0.0419
Epoch 16/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.5198 - acc: 0.
0514  ETA: 10s - loss:  - ET - ETA - ETA:Epoch 00015: val_loss improved from
4.61243 to 4.60220, saving model to saved_models/weights.best.from_scratch.hd
f5
6680/6680 [==============================] - 11s - loss: 4.5196 - acc: 0.0513
- val_loss: 4.6022 - val_acc: 0.0299
Epoch 17/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.4939 - acc: 0.
0536Epoch 00016: val_loss improved from 4.60220 to 4.57842, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 11s - loss: 4.4940 - acc: 0.0536
- val_loss: 4.5784 - val_acc: 0.0311
Epoch 18/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.4711 - acc: 0.
0541Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.4714 - acc: 0.0542
- val_loss: 4.5851 - val_acc: 0.0335
Epoch 19/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.4507 - acc: 0.
0583Epoch 00018: val_loss improved from 4.57842 to 4.54899, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.4508 - acc: 0.0582
- val_loss: 4.5490 - val_acc: 0.0383
Epoch 20/20
```

```
6660/6680 [============================>.] - ETA: 0s - loss: 4.4328 - acc: 0.
0599Epoch 00019: val_loss improved from 4.54899 to 4.53483, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.4336 - acc: 0.0597
- val_loss: 4.5348 - val_acc: 0.0359
```

Out[15]:
```
<keras.callbacks.History at 0x7ff3bb9819e8>
```

Load the Model with the Best Validation Loss

In [16]:
```python
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

Test the Model

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

In [17]:
```python
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0
))) for tensor in test_tensors]

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

```
Test accuracy: 6.1005%
```

Step 4: Use a CNN to Classify Dog Breeds

To reduce training time without sacrificing accuracy, we show you how to train a CNN using transfer learning. In the following step, you will get a chance to use transfer learning to train your own CNN.

Obtain Bottleneck Features

In [18]:
```python
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

In [19]:
```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

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

Compile the Model

In [20]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metr
ics=['accuracy'])
```

Train the Model

In [21]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5'
,
                               verbose=1, save_best_only=True)

VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6640/6680 [============================>.] - ETA: 0s - loss: 12.2870 - ac
c: 0.1261Epoch 00000: val_loss improved from inf to 10.94347, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s - loss: 12.2738 - acc: 0.1
265 - val_loss: 10.9435 - val_acc: 0.2060
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 10.1586 - ac
c: 0.2778Epoch 00001: val_loss improved from 10.94347 to 10.09595, saving
model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.1583 - acc: 0.2
780 - val_loss: 10.0959 - val_acc: 0.2611
Epoch 3/20
6580/6680 [============================>.] - ETA: 0s - loss: 9.5715 - acc:
0.3477Epoch 00002: val_loss improved from 10.09595 to 9.76786, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5872 - acc: 0.34
70 - val_loss: 9.7679 - val_acc: 0.3126
Epoch 4/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.3478 - acc:
0.3761Epoch 00003: val_loss improved from 9.76786 to 9.69208, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.3585 - acc: 0.37
53 - val_loss: 9.6921 - val_acc: 0.3198
Epoch 5/20
6620/6680 [============================>.] - ETA: 0s - loss: 9.2488 - acc:
0.3940Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.2435 - acc: 0.39
45 - val_loss: 9.6921 - val_acc: 0.3317
Epoch 6/20
6580/6680 [============================>.] - ETA: 0s - loss: 9.1026 - acc:
0.4103Epoch 00005: val_loss improved from 9.69208 to 9.54106, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1223 - acc: 0.40
90 - val_loss: 9.5411 - val_acc: 0.3305
Epoch 7/20
6600/6680 [============================>.] - ETA: 0s - loss: 9.0053 - acc:
0.4180Epoch 00006: val_loss improved from 9.54106 to 9.40061, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.9951 - acc: 0.41
89 - val_loss: 9.4006 - val_acc: 0.3437
Epoch 8/20
6580/6680 [============================>.] - ETA: 0s - loss: 8.8633 - acc:
0.4274Epoch 00007: val_loss improved from 9.40061 to 9.36055, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.8551 - acc: 0.42
78 - val_loss: 9.3605 - val_acc: 0.3545
Epoch 9/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.7424 - acc:
0.4398Epoch 00008: val_loss improved from 9.36055 to 9.24619, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7343 - acc: 0.44
01 - val_loss: 9.2462 - val_acc: 0.3533
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.5745 - acc:
0.4521Epoch 00009: val_loss improved from 9.24619 to 9.16664, saving model
```

```
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5730 - acc: 0.45
22 - val_loss: 9.1666 - val_acc: 0.3713
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.4019 - acc:
0.4605Epoch 00010: val_loss improved from 9.16664 to 8.92184, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.4033 - acc: 0.46
05 - val_loss: 8.9218 - val_acc: 0.3749
Epoch 12/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.2506 - acc:
0.4750Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.2641 - acc: 0.47
43 - val_loss: 8.9386 - val_acc: 0.3689
Epoch 13/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.1824 - acc:
0.4787Epoch 00012: val_loss improved from 8.92184 to 8.74246, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s - loss: 8.1786 - acc: 0.47
90 - val_loss: 8.7425 - val_acc: 0.3952
Epoch 14/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.0973 - acc:
0.4871Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.0724 - acc: 0.48
89 - val_loss: 8.7455 - val_acc: 0.3952
Epoch 15/20
6640/6680 [============================>.] - ETA: 0s - loss: 8.0159 - acc:
0.4908Epoch 00014: val_loss improved from 8.74246 to 8.71035, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0119 - acc: 0.49
10 - val_loss: 8.7104 - val_acc: 0.3904
Epoch 16/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.9659 - acc:
0.4986Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 7.9355 - acc: 0.50
06 - val_loss: 8.7481 - val_acc: 0.3880
Epoch 17/20
6540/6680 [============================>.] - ETA: 0s - loss: 7.8942 - acc:
0.5035Epoch 00016: val_loss improved from 8.71035 to 8.58572, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.9105 - acc: 0.50
22 - val_loss: 8.5857 - val_acc: 0.4120
Epoch 18/20
6540/6680 [============================>.] - ETA: 0s - loss: 7.8329 - acc:
0.5035Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 7.8354 - acc: 0.50
34 - val_loss: 8.6455 - val_acc: 0.3964
Epoch 19/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.7891 - acc:
0.5089Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 7.7820 - acc: 0.50
94 - val_loss: 8.5891 - val_acc: 0.3952
Epoch 20/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.7585 - acc:
0.5105Epoch 00019: val_loss improved from 8.58572 to 8.56961, saving model
to saved_models/weights.best.VGG16.hdf5
```

```
6680/6680 [==============================] - 1s - loss: 7.7618 - acc: 0.51
03 - val_loss: 8.5696 - val_acc: 0.3988
```

Out[21]:
```
<keras.callbacks.History at 0x7ff3bb6e3898>
```

Load the Model with the Best Validation Loss

In [22]:
```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

Test the Model

Now, we can use the CNN to test how well it identifies breed within our test dataset of dog images. We print the test accuracy below.

In [23]:
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axi
s=0))) for feature in test_VGG16]

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

```
Test accuracy: 40.9091%
```

Predict Dog Breed with the Model

In [24]:
```
from extract_bottleneck_features import *

def VGG16_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_VGG16(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG16_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

In Step 4, we used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, you must use the bottleneck features from a different pre-trained model. To make things easier for you, we have pre-computed the features for all of the networks that are currently available in Keras:

- VGG-19 (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz) bottleneck features
- ResNet-50 (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz) bottleneck features
- Inception (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz) bottleneck features
- Xception (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogXceptionData.npz) bottleneck features

The files are encoded as such:

```
Dog{network}Data.npz
```

where `{network}`, in the above filename, can be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`. Pick one of the above architectures, download the corresponding bottleneck features, and store the downloaded file in the `bottleneck_features/` folder in the repository.

(IMPLEMENTATION) Obtain Bottleneck Features

In the code block below, extract the bottleneck features corresponding to the train, test, and validation sets by running the following:

```
bottleneck_features = np.load('bottleneck_features/Dog{network}Data.npz')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

In [25]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features = np.load('bottleneck_features/DogVGG19Data.npz')
train_VGG19 = bottleneck_features['train']
valid_VGG19 = bottleneck_features['valid']
test_VGG19 = bottleneck_features['test']

bottleneck_features = np.load('bottleneck_features/DogResnet50Data.npz')
train_Resnet50 = bottleneck_features['train']
valid_Resnet50 = bottleneck_features['valid']
test_Resnet50 = bottleneck_features['test']

bottleneck_features = np.load('bottleneck_features/DogInceptionV3Data.npz')
train_InceptionV3 = bottleneck_features['train']
valid_InceptionV3 = bottleneck_features['valid']
test_InceptionV3 = bottleneck_features['test']

bottleneck_features = np.load('bottleneck_features/DogXceptionData.npz')
train_Xception = bottleneck_features['train']
valid_Xception = bottleneck_features['valid']
test_Xception = 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: Added Desnse and Dropout layers and this increased the test accuracy. If optimiser with different parameters and epoches, the test accuracy is improved and saw more than 85% is some cases. The earlier CNNs does not use middle layer Dense and Drop layer where as this model uses and best suits for this dog breeder problem.

In [26]:
```
### TODO: Define your architecture.
VGG19_model = Sequential()
VGG19_model.add(GlobalAveragePooling2D(input_shape=train_VGG19.shape[1:]))
VGG19_model.add(Dense(256, activation='relu'))
VGG19_model.add(Dropout(0.3))
VGG19_model.add(Dense(133, activation='softmax'))
VGG19_model.summary()

Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[
1:]))
Resnet50_model.add(Dense(256, activation='relu'))
Resnet50_model.add(Dropout(0.3))
Resnet50_model.add(Dense(133, activation='softmax'))
Resnet50_model.summary()

InceptionV3_model = Sequential()
InceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.
shape[1:]))
InceptionV3_model.add(Dense(256, activation='relu'))
InceptionV3_model.add(Dropout(0.3))
InceptionV3_model.add(Dense(133, activation='softmax'))
InceptionV3_model.summary()

Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[
1:]))
Xception_model.add(Dense(256, activation='relu'))
Xception_model.add(Dropout(0.3))
Xception_model.add(Dense(133, activation='softmax'))
Xception_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_3 ( (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 256)               131328    
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 133)               34181     
=================================================================
Total params: 165,509.0
Trainable params: 165,509.0
Non-trainable params: 0.0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_4 ( (None, 2048)              0         
_________________________________________________________________
dense_5 (Dense)              (None, 256)               524544    
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 133)               34181     
=================================================================
Total params: 558,725.0
Trainable params: 558,725.0
Non-trainable params: 0.0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_5 ( (None, 2048)              0         
_________________________________________________________________
dense_7 (Dense)              (None, 256)               524544    
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_8 (Dense)              (None, 133)               34181     
=================================================================
Total params: 558,725.0
Trainable params: 558,725.0
Non-trainable params: 0.0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_6 ( (None, 2048)              0         
_________________________________________________________________
dense_9 (Dense)              (None, 256)               524544    
_________________________________________________________________
dropout_4 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 133)               34181     
=================================================================
Total params: 558,725.0
```

```
Trainable params: 558,725.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [27]:
```
### TODO: Compile the model.
from keras import optimizers
rms = optimizers.rmsprop()
sgd = optimizers.SGD()
VGG19_model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=[
'accuracy'])
Resnet50_model.compile(loss='categorical_crossentropy', optimizer=rms, metrics
=['accuracy'])
InceptionV3_model.compile(loss='categorical_crossentropy', optimizer=sgd, metr
ics=['accuracy'])
Xception_model.compile(loss='categorical_crossentropy', optimizer=rms, 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)](https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

In [28]:
```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG19.hd
f5',
                               verbose=1, save_best_only=True)
VGG19_model.fit(train_VGG19, train_targets,
          validation_data=(valid_VGG19, valid_targets),
          epochs=10, batch_size=20, callbacks=[checkpointer], verbose=1)

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.RESNET5
0.hdf5',
                                verbose=1, save_best_only=True)
Resnet50_model.fit(train_Resnet50, train_targets,
           validation_data=(valid_Resnet50, valid_targets),
           epochs=10, batch_size=20, callbacks=[checkpointer], verbose=1)

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.INCEPTIO
N.hdf5',
                                verbose=1, save_best_only=True)
InceptionV3_model.fit(train_InceptionV3, train_targets,
           validation_data=(valid_InceptionV3, valid_targets),
           epochs=10, batch_size=20, callbacks=[checkpointer], verbose=1)

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.XCEPTIO
N.hdf5',
                               verbose=1, save_best_only=True)
Xception_model.fit(train_Xception, train_targets,
          validation_data=(valid_Xception, valid_targets),
          epochs=10, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.0856 - acc:
0.2381Epoch 00000: val_loss improved from inf to 1.92134, saving model to
saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 4.0807 - acc: 0.23
89 - val_loss: 1.9213 - val_acc: 0.5150
Epoch 2/10
6560/6680 [============================>.] - ETA: 0s - loss: 1.8698 - acc:
0.5110Epoch 00001: val_loss improved from 1.92134 to 1.25680, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 1.8695 - acc: 0.51
08 - val_loss: 1.2568 - val_acc: 0.6407
Epoch 3/10
6520/6680 [============================>.] - ETA: 0s - loss: 1.3528 - acc:
0.6216Epoch 00002: val_loss improved from 1.25680 to 1.06632, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 1.3549 - acc: 0.62
20 - val_loss: 1.0663 - val_acc: 0.6778
Epoch 4/10
6660/6680 [============================>.] - ETA: 0s - loss: 1.0540 - acc:
0.6992Epoch 00003: val_loss improved from 1.06632 to 0.98661, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 1.0524 - acc: 0.69
97 - val_loss: 0.9866 - val_acc: 0.6934
Epoch 5/10
6500/6680 [============================>.] - ETA: 0s - loss: 0.8832 - acc:
0.7328Epoch 00004: val_loss improved from 0.98661 to 0.92880, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 0.8843 - acc: 0.73
25 - val_loss: 0.9288 - val_acc: 0.7222
Epoch 6/10
6560/6680 [============================>.] - ETA: 0s - loss: 0.7999 - acc:
0.7555Epoch 00005: val_loss improved from 0.92880 to 0.91065, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 0.8011 - acc: 0.75
52 - val_loss: 0.9107 - val_acc: 0.7317
Epoch 7/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.6960 - acc:
0.7850Epoch 00006: val_loss improved from 0.91065 to 0.86108, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 0.6958 - acc: 0.78
50 - val_loss: 0.8611 - val_acc: 0.7413
Epoch 8/10
6580/6680 [============================>.] - ETA: 0s - loss: 0.6213 - acc:
0.8103Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 2s - loss: 0.6233 - acc: 0.80
91 - val_loss: 0.8616 - val_acc: 0.7329
Epoch 9/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.5540 - acc:
0.8266Epoch 00008: val_loss improved from 0.86108 to 0.84712, saving model
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 0.5539 - acc: 0.82
66 - val_loss: 0.8471 - val_acc: 0.7497
Epoch 10/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.4974 - acc:
0.8375Epoch 00009: val_loss improved from 0.84712 to 0.81343, saving model
```

```
to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 0.4987 - acc: 0.83
74 - val_loss: 0.8134 - val_acc: 0.7545
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6560/6680 [============================>.] - ETA: 0s - loss: 2.4475 - acc:
0.4160Epoch 00000: val_loss improved from inf to 0.98958, saving model to
saved_models/weights.best.RESNET50.hdf5
6680/6680 [==============================] - 2s - loss: 2.4238 - acc: 0.42
04 - val_loss: 0.9896 - val_acc: 0.6958
Epoch 2/10
6640/6680 [============================>.] - ETA: 0s - loss: 0.9431 - acc:
0.7142Epoch 00001: val_loss improved from 0.98958 to 0.74691, saving model
to saved_models/weights.best.RESNET50.hdf5
6680/6680 [==============================] - 1s - loss: 0.9471 - acc: 0.71
33 - val_loss: 0.7469 - val_acc: 0.7772
Epoch 3/10
6500/6680 [============================>.] - ETA: 0s - loss: 0.6398 - acc:
0.7992Epoch 00002: val_loss improved from 0.74691 to 0.65658, saving model
to saved_models/weights.best.RESNET50.hdf5
6680/6680 [==============================] - 1s - loss: 0.6396 - acc: 0.79
90 - val_loss: 0.6566 - val_acc: 0.8036
Epoch 4/10
6580/6680 [============================>.] - ETA: 0s - loss: 0.4890 - acc:
0.8426Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.4879 - acc: 0.84
31 - val_loss: 0.7319 - val_acc: 0.7916
Epoch 5/10
6560/6680 [============================>.] - ETA: 0s - loss: 0.3929 - acc:
0.8712Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3903 - acc: 0.87
17 - val_loss: 0.7178 - val_acc: 0.7928
Epoch 6/10
6580/6680 [============================>.] - ETA: 0s - loss: 0.3372 - acc:
0.8891Epoch 00005: val_loss improved from 0.65658 to 0.64144, saving model
to saved_models/weights.best.RESNET50.hdf5
6680/6680 [==============================] - 1s - loss: 0.3371 - acc: 0.88
89 - val_loss: 0.6414 - val_acc: 0.8204
Epoch 7/10
6560/6680 [============================>.] - ETA: 0s - loss: 0.2978 - acc:
0.9009Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2963 - acc: 0.90
16 - val_loss: 0.7153 - val_acc: 0.8072
Epoch 8/10
6500/6680 [============================>.] - ETA: 0s - loss: 0.2571 - acc:
0.9182Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2582 - acc: 0.91
80 - val_loss: 0.8301 - val_acc: 0.7928
Epoch 9/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.2305 - acc:
0.9261Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2311 - acc: 0.92
57 - val_loss: 0.7655 - val_acc: 0.8216
Epoch 10/10
6600/6680 [============================>.] - ETA: 0s - loss: 0.2026 - acc:
0.9323Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2036 - acc: 0.93
```

```
17 - val_loss: 0.8002 - val_acc: 0.8192
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6580/6680 [============================>.] - ETA: 0s - loss: 3.2372 - acc:
0.3728Epoch 00000: val_loss improved from inf to 1.41007, saving model to
saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 3.2152 - acc: 0.37
65 - val_loss: 1.4101 - val_acc: 0.7329
Epoch 2/10
6620/6680 [============================>.] - ETA: 0s - loss: 1.2575 - acc:
0.7045Epoch 00001: val_loss improved from 1.41007 to 0.75930, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 1.2529 - acc: 0.70
52 - val_loss: 0.7593 - val_acc: 0.7940
Epoch 3/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.8591 - acc:
0.7722Epoch 00002: val_loss improved from 0.75930 to 0.60453, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.8590 - acc: 0.77
17 - val_loss: 0.6045 - val_acc: 0.8240
Epoch 4/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.6994 - acc:
0.8065Epoch 00003: val_loss improved from 0.60453 to 0.56861, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.7023 - acc: 0.80
57 - val_loss: 0.5686 - val_acc: 0.8287
Epoch 5/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.5963 - acc:
0.8313Epoch 00004: val_loss improved from 0.56861 to 0.53787, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.5965 - acc: 0.83
16 - val_loss: 0.5379 - val_acc: 0.8347
Epoch 6/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.5352 - acc:
0.8443Epoch 00005: val_loss improved from 0.53787 to 0.52579, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.5356 - acc: 0.84
40 - val_loss: 0.5258 - val_acc: 0.8383
Epoch 7/10
6580/6680 [============================>.] - ETA: 0s - loss: 0.4947 - acc:
0.8498Epoch 00006: val_loss improved from 0.52579 to 0.50168, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.4956 - acc: 0.84
99 - val_loss: 0.5017 - val_acc: 0.8407
Epoch 8/10
6600/6680 [============================>.] - ETA: 0s - loss: 0.4377 - acc:
0.8720Epoch 00007: val_loss improved from 0.50168 to 0.48810, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.4375 - acc: 0.87
19 - val_loss: 0.4881 - val_acc: 0.8455
Epoch 9/10
6620/6680 [============================>.] - ETA: 0s - loss: 0.4208 - acc:
0.8731Epoch 00008: val_loss improved from 0.48810 to 0.48426, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.4204 - acc: 0.87
35 - val_loss: 0.4843 - val_acc: 0.8431
Epoch 10/10
```

```
6620/6680 [============================>.] - ETA: 0s - loss: 0.3891 - acc:
0.8813Epoch 00009: val_loss improved from 0.48426 to 0.48125, saving model
to saved_models/weights.best.INCEPTION.hdf5
6680/6680 [==============================] - 3s - loss: 0.3891 - acc: 0.88
10 - val_loss: 0.4812 - val_acc: 0.8551
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6660/6680 [============================>.] - ETA: 0s - loss: 1.4708 - acc:
0.6444Epoch 00000: val_loss improved from inf to 0.60715, saving model to
saved_models/weights.best.XCEPTION.hdf5
6680/6680 [==============================] - 5s - loss: 1.4676 - acc: 0.64
54 - val_loss: 0.6072 - val_acc: 0.8180
Epoch 2/10
6600/6680 [============================>.] - ETA: 0s - loss: 0.6043 - acc:
0.8124Epoch 00001: val_loss improved from 0.60715 to 0.59133, saving model
to saved_models/weights.best.XCEPTION.hdf5
6680/6680 [==============================] - 4s - loss: 0.6057 - acc: 0.81
20 - val_loss: 0.5913 - val_acc: 0.8096
Epoch 3/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.4705 - acc:
0.8508Epoch 00002: val_loss improved from 0.59133 to 0.58875, saving model
to saved_models/weights.best.XCEPTION.hdf5
6680/6680 [==============================] - 4s - loss: 0.4716 - acc: 0.85
03 - val_loss: 0.5888 - val_acc: 0.8204
Epoch 4/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.3889 - acc:
0.8725Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 4s - loss: 0.3887 - acc: 0.87
28 - val_loss: 0.5996 - val_acc: 0.8204
Epoch 5/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.3275 - acc:
0.8920Epoch 00004: val_loss improved from 0.58875 to 0.55763, saving model
to saved_models/weights.best.XCEPTION.hdf5
6680/6680 [==============================] - 4s - loss: 0.3269 - acc: 0.89
22 - val_loss: 0.5576 - val_acc: 0.8311
Epoch 6/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.2768 - acc:
0.9072Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 5s - loss: 0.2762 - acc: 0.90
75 - val_loss: 0.6530 - val_acc: 0.8407
Epoch 7/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.2478 - acc:
0.9152Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 5s - loss: 0.2483 - acc: 0.91
50 - val_loss: 0.5754 - val_acc: 0.8347
Epoch 8/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.2180 - acc:
0.9239Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 5s - loss: 0.2178 - acc: 0.92
38 - val_loss: 0.6076 - val_acc: 0.8419
Epoch 9/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.1871 - acc:
0.9389Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 5s - loss: 0.1866 - acc: 0.93
91 - val_loss: 0.6629 - val_acc: 0.8491
Epoch 10/10
6660/6680 [============================>.] - ETA: 0s - loss: 0.1749 - acc:
```

```
0.9414Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 5s - loss: 0.1746 - acc: 0.94
15 - val_loss: 0.6935 - val_acc: 0.8287
```

Out[28]:

```
<keras.callbacks.History at 0x7ff3bac8ef60>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [29]:

```
### TODO: Load the model weights with the best validation loss.
VGG19_model.load_weights('saved_models/weights.best.VGG19.hdf5')
Resnet50_model.load_weights('saved_models/weights.best.RESNET50.hdf5')
InceptionV3_model.load_weights('saved_models/weights.best.INCEPTION.hdf5')
Xception_model.load_weights('saved_models/weights.best.XCEPTION.hdf5')
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

In [30]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
VGG19_predictions = [np.argmax(VGG19_model.predict(np.expand_dims(feature, axi
s=0))) for feature in test_VGG19]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG19_predictions)==np.argmax(test_targets
, axis=1))/len(VGG19_predictions)
print('Test accuracy of VGG-19: %.4f%%' % test_accuracy)

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

# report test accuracy
test_accuracy = 100*np.sum(np.array(Resnet50_predictions)==np.argmax(test_targ
ets, axis=1))/len(Resnet50_predictions)
print('Test accuracy of Resnet-50: %.4f%%' % test_accuracy)

InceptionV3_predictions = [np.argmax(InceptionV3_model.predict(np.expand_dims(
feature, axis=0))) for feature in test_InceptionV3]

# report test accuracy
test_accuracy = 100*np.sum(np.array(InceptionV3_predictions)==np.argmax(test_t
argets, axis=1))/len(InceptionV3_predictions)
print('Test accuracy of Inception V3: %.4f%%' % test_accuracy)

Xception_predictions = [np.argmax(Xception_model.predict(np.expand_dims(featur
e, axis=0))) for feature in test_Xception]

# report test accuracy
test_accuracy = 100*np.sum(np.array(Xception_predictions)==np.argmax(test_targ
ets, axis=1))/len(Xception_predictions)
print('Test accuracy of Xception: %.4f%%' % test_accuracy)
```

```
Test accuracy of VGG-19: 75.5981%
Test accuracy of Resnet-50: 82.2967%
Test accuracy of Inception V3: 81.5789%
Test accuracy of Xception: 83.2536%
```

(IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan_hound`, etc) that is predicted by your model.

Similar to the analogous function in Step 5, your function should have three steps:

1. Extract the bottleneck features corresponding to the chosen CNN model.
2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed.
3. Use the `dog_names` array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in `extract_bottleneck_features.py`, and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

```
extract_{network}
```

where `{network}`, in the above filename, should be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`.

In [31]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
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)]

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

def InceptionV3_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_InceptionV3(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = InceptionV3_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]

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

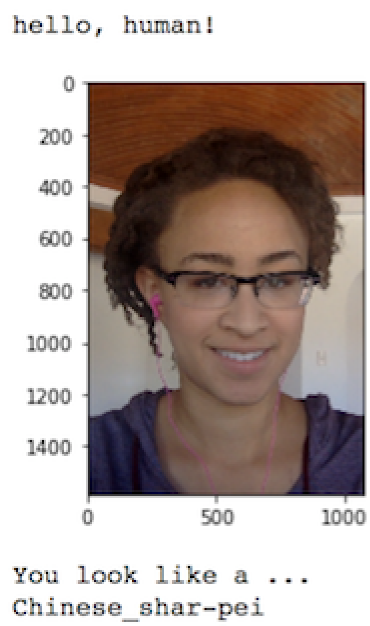
Step 6: Write your Algorithm

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

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

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

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



(IMPLEMENTATION) Write your Algorithm

In [32]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
%matplotlib inline

def verify(image_path):
    human = face_detector(image_path)
    dog = dog_detector(image_path)

    if human > 0:
        print("hello, human!")

    if dog > 0:
        print("hello, dog!")

    # disply the test image
    img = cv2.imread(image_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()

    if dog > 0 or human > 0:
        print("You look like a ... ")
        result = VGG19_predict_breed(image_path)
        print(result, 'in VGG19 prediction model.')
        result = Resnet50_predict_breed(image_path)
        print(result, 'in RESNET50 prediction model.')
        result = InceptionV3_predict_breed(image_path)
        print(result, 'in INCEPTIONV3 prediction model.')
        result = Xception_predict_breed(image_path)
        print(result, 'in XCEPTION prediction model.')
        return

    if dog == 0 or human == 0:
        print("Neither dog or human, it's not a correct input image.")

        return
```

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.

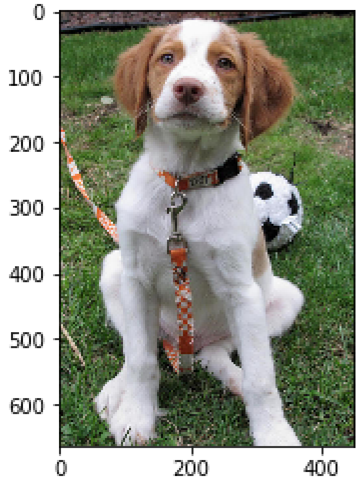
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: The output is not so good and not so worse. In the case of American_water_spaniel_00648.jpg, two prediction models gave the correct and another not. For the case of human, each prediction model gave different dog breed, which is acceptable. Following are three improvements:

1. Define function for each of the prediction model, where lot of code will be re-used instead of duplicating the code.
2. Pass different parameters of optimers to this function with different optimer parameters like learning rate, decay etc, which can help for over fitting problem. Consider only the best accuracy model.
3. Add another Dense layer with 512 and drop out layer in addition to the existing CNN and check test accuracy.

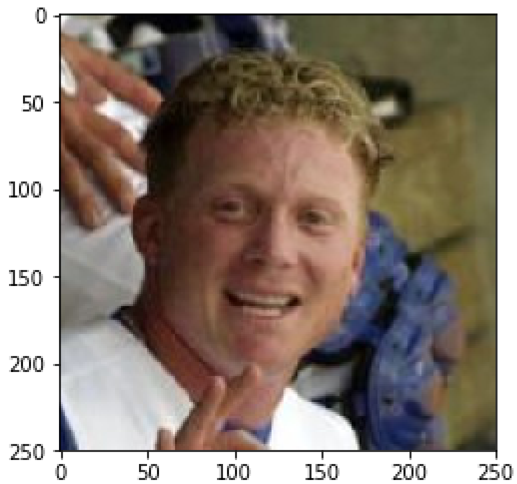
In [33]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

#verify('images/Labrador_retriever_06449.jpg')

test_images = np.array(glob("test_images/*"))

# print statistics about the dataset
print('There are %d total test images.' % len(test_images))

#for test_image in test_images:
for index in range(len(test_images)):
    print('Index =', index, ',test image is', test_images[index])
    verify(test_images[index])
```

```
There are 6 total test images.
Index = 0 ,test image is test_images/Brittany_02625.jpg
hello, dog!
```

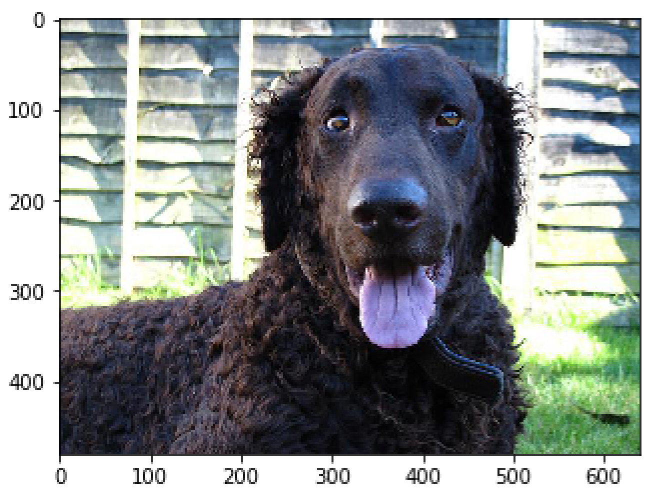



```
You look like a ...
Brittany in VGG19 prediction model.
Brittany in RESNET50 prediction model.
Brittany in INCEPTIONV3 prediction model.
Brittany in XCEPTION prediction model.
Index = 1 ,test image is test_images/American_water_spaniel_00648.jpg
hello, dog!
```

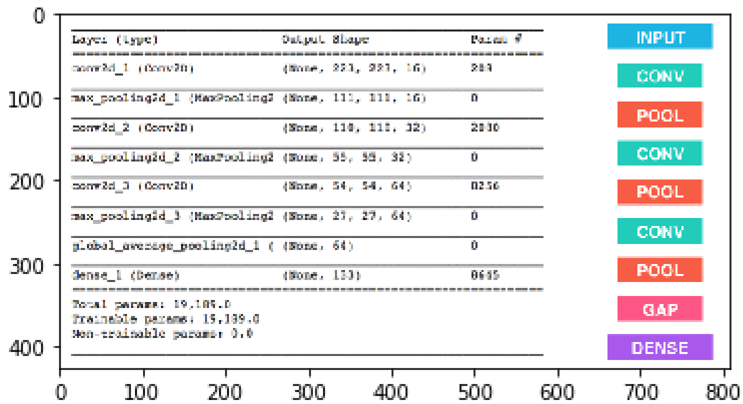

```
You look like a ...
Irish_water_spaniel in VGG19 prediction model.
American_water_spaniel in RESNET50 prediction model.
American_water_spaniel in INCEPTIONV3 prediction model.
Curly-coated_retriever in XCEPTION prediction model.
Index = 2 ,test image is test_images/Aaron_Guiel_0001.jpg
hello, human!
```



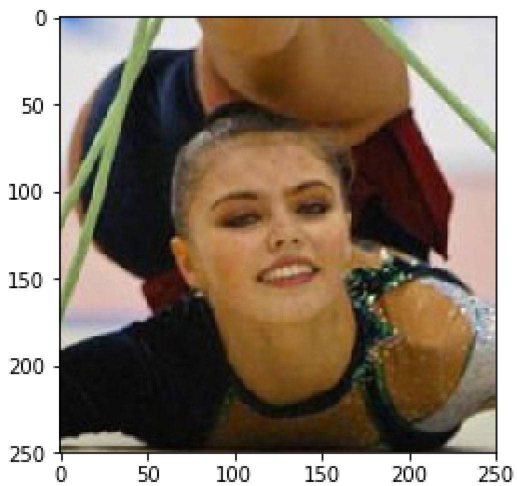
```
You look like a ...
Brittany in VGG19 prediction model.
Australian_cattle_dog in RESNET50 prediction model.
Wirehaired_pointing_griffon in INCEPTIONV3 prediction model.
Dachshund in XCEPTION prediction model.
Index = 3 ,test image is test_images/Curly-coated_retriever_03896.jpg
hello, dog!
```



```
You look like a ...
Curly-coated_retriever in VGG19 prediction model.
Curly-coated_retriever in RESNET50 prediction model.
Curly-coated_retriever in INCEPTIONV3 prediction model.
Curly-coated_retriever in XCEPTION prediction model.
Index = 4 ,test image is test_images/sample_cnn.png
```




```
Neither dog or human, it's not a correct input image.
Index = 5 ,test image is test_images/Alina_Kabaeva_0001.jpg
hello, human!
```



```
You look like a ...
Pointer in VGG19 prediction model.
American_staffordshire_terrier in RESNET50 prediction model.
Greyhound in INCEPTIONV3 prediction model.
Dachshund in XCEPTION prediction model.
```