Artificial Intelligence Nanodegree

Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

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

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

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

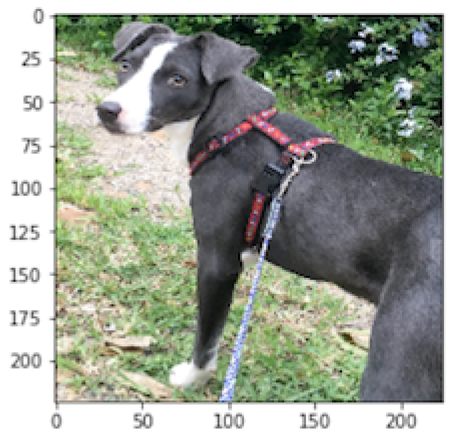
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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

Why We're Here

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

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



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

The Road Ahead

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

Step 0: Import Datasets

Import Dog Dataset

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

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

In [1]:

```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob

# define function to load train, test, and validation datasets
def load_dataset(path):
    data = load_files(path)
    dog_files = np.array(data['filenames'])
    dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
    return dog_files, dog_targets

# load train, test, and validation datasets
train_files, train_targets = load_dataset('dogImages/train')
valid_files, valid_targets = load_dataset('dogImages/valid')
test_files, test_targets = load_dataset('dogImages/test')

# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*/"))]

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_files, test_fi
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

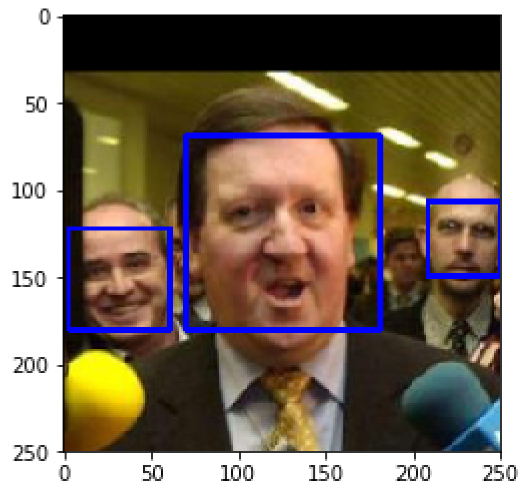
We use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades).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

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

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

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

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

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

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

```
Number of faces detected: 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:

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.
counter1 = 0
counter2 = 0

for img in human_files_short:
    if(face_detector(img)):
        counter1 = counter1 + 1
per1 = (counter1 * 100.0)/100
print('Human files predection percentage = ', per1)


for img in dog_files_short:
    if(face_detector(img)):
        counter2 = counter2 + 1
per2 = (counter2 * 100.0)/100
print('Dog files predection percentage = ', per2)
```

```
Human files predection percentage =  99.0
Dog files predection percentage =  11.0
```

Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having unneccessarily frustrated users!). In your opinion, is this a reasonable expectation to pose on the user? If not, can you think of a way to detect humans in images that does not necessitate an image with a clearly presented face?

Answer: As from the above function it is predecting 99% accurately on the human images and only small case is wrongly detected. I guess the images entirely not visible are detected as dogs. We can train deep learning algorithm to differentiate those features too. Like for a given image whether it is dog or 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 [ ]:

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

Step 2: Detect Dogs

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

1000 categories (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a). Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

In [6]:

```
from keras.applications.resnet50 import ResNet50

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

```
Downloading data from https://github.com/fchollet/deep-learning-models/relea
ses/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels.h5 (https://gi
thub.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weigh
ts_tf_dim_ordering_tf_kernels.h5)
```

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 [7]:

```
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 [8]:

```
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 [9]:
```

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

(IMPLEMENTATION) Assess the Dog Detector

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

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

Answer:

```
In [10]:
```

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
counter1 = 0
counter2 = 0

for img in human_files_short:
    if(dog_detector(img)):
        counter1 = counter1 + 1
per1 = (counter1 * 100.0)/100
print('Human files predection percentage = ', per1)


for img in dog_files_short:
    if(dog_detector(img)):
        counter2 = counter2 + 1
per2 = (counter2 * 100.0)/100
print('Dog files predection percentage = ', per2)
```

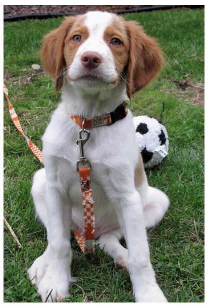
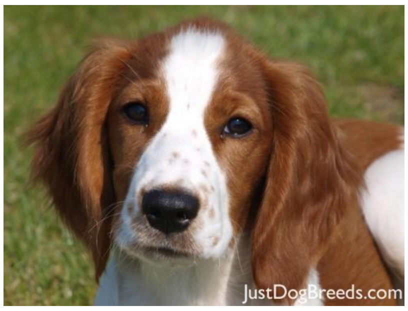
```
Human files predection percentage =  0.0
Dog files predection percentage =  100.0
```

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

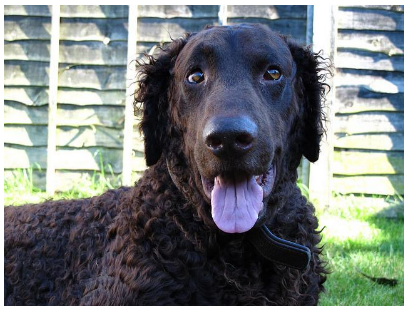
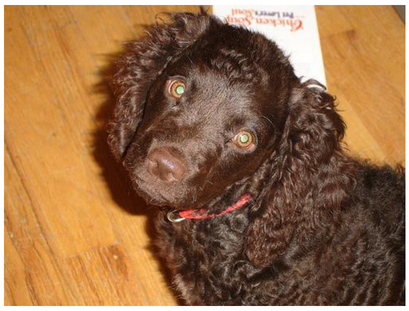
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

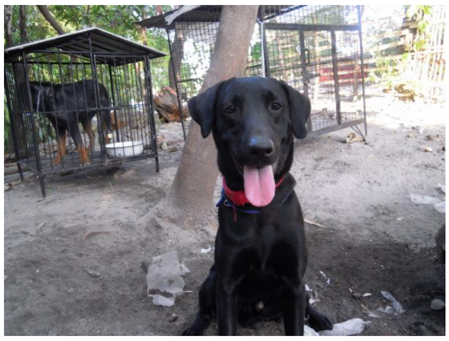
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

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 [11]:

```python
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
```

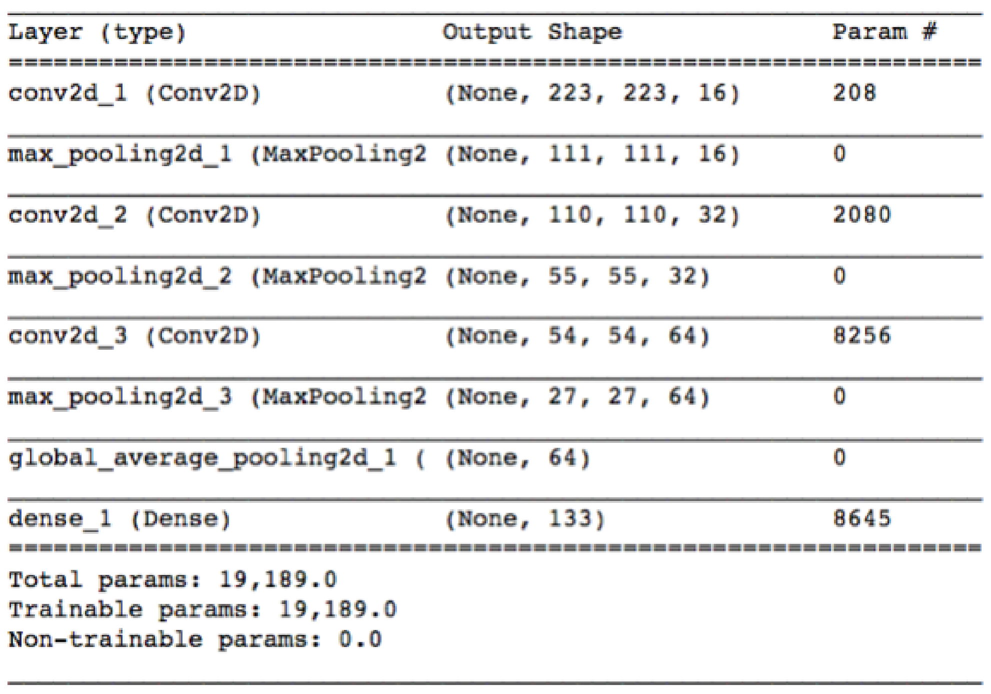
```
100%|██████████| 6680/6680 [00:53<00:00, 124.19it/s]
100%|██████████| 835/835 [00:06<00:00, 138.39it/s]
100%|██████████| 836/836 [00:06<00:00, 138.65it/s]
```

(IMPLEMENTATION) Model Architecture

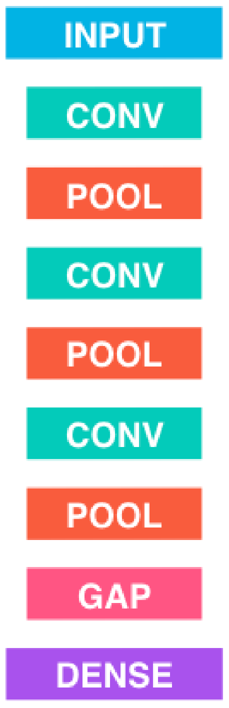
Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
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: Convolutional neural network is a combination of convolutional map, filters, max pooling layers with activation and dropout functions.

For a given input image having dimensions (1,224,224,3)

- First we need to define filters to run through entire image for capturing different features of image

- each filter slide through entire image with activation function like Relu(rectified linear unit) produce a convolutional map
- Again we can apply same filters to better capture different featues and produce a another convolutional map
- Although the role of the convolutional layer is to detect local conjunctions of features from the previous layer, the role of the pooling layer is to merge semantically similar features into one. Because the relative positions of the features forming a motif can vary somewhat, reliably detecting the motif can be done by coarse-graining the position of each feature. A typical pooling unit computes the maximum of a local patch of units in one feature map (or in a few feature maps).Neighbouring pooling units take input from patches that are shifted by more than one row or column, thereby reducing the dimension of the representation and creating an invariance to small shifts and distortions. Two or three stages of convolution, non-linearity and pooling are stacked, followed by more convolutional and fully-connected layers.

In our case the model have:

- We are applying 32 filters with each having size of (3,3) on input image having dimension (1,224,224,3) producing 32 convolutional maps having size of (1,222,222,3)
- Applying maxpool will reduce given convolutional map to half of its size. the second layer having 32 feature or convolutional maps having size of (1,111,111,3)
- We are again applying 64 filters with each having size of (3,3) on input previous convolutional producing 64 convolutional maps having size of (1,111,111,3)
- We can follow same pattern upto three iterations and then flat the model and apply dense layer to convert the model into classify our image classes.
- Flatten layer will flatten entire layer into single vector. And dense layer reshape it into multi row, single column matrix for classifying our image data.
- Our final model is a fully connected dense layer with softmax(categorical) activation to classify given image to one of 133 classes.
- I hope above model is basic step to improve further to get better predictions.

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(32, kernel_size=(3, 3),activation='relu',input_shape=(224,224,3)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(133, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 222, 222, 32)      896       
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 111, 111, 32)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 111, 111, 64)      18496     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 55, 55, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 55, 55, 128)       73856     
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 27, 27, 128)       0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 27, 27, 128)       0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 93312)             0         
_________________________________________________________________
dense_1 (Dense)              (None, 250)               23328250  
_________________________________________________________________
dropout_2 (Dropout)          (None, 250)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               33383     
=================================================================
Total params: 23,454,881
Trainable params: 23,454,881
Non-trainable params: 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 = 50

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

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

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.9271 - acc:
0.0093Epoch 00000: val_loss improved from inf to 4.87246, saving model to
 saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 49s - loss: 4.9270 - acc: 0.0
093 - val_loss: 4.8725 - val_acc: 0.0084
Epoch 2/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.8042 - acc:
0.0198Epoch 00001: val_loss improved from 4.87246 to 4.61660, saving model
to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 49s - loss: 4.8037 - acc: 0.0
199 - val_loss: 4.6166 - val_acc: 0.0395
Epoch 3/50
6660/6680 [============================>.] - ETA: 0s - loss: 4.4924 - acc:
0.0395Epoch 00002: val_loss improved from 4.61660 to 4.31463, saving model
to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 49s - loss: 4.4920 - acc: 0.0
394 - val_loss: 4.3146 - val_acc: 0.0743
```

Load the Model with the Best Validation Loss

In [16]:

```
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]:

```
# 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 tensc

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

```
Test accuracy: 9.8086%
```

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]:

```
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_1 ( (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [20]:

```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accurac
```

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
6580/6680 [============================>.] - ETA: 0s - loss: 12.8252 - acc:
 0.1065Epoch 00000: val_loss improved from inf to 11.28914, saving model to
 saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 12.8068 - acc: 0.108
5 - val_loss: 11.2891 - val_acc: 0.1868
Epoch 2/20
6460/6680 [============================>.] - ETA: 0s - loss: 10.7373 - acc:
 0.2515Epoch 00001: val_loss improved from 11.28914 to 10.62197, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.7303 - acc: 0.252
1 - val_loss: 10.6220 - val_acc: 0.2587
Epoch 3/20
6600/6680 [============================>.] - ETA: 0s - loss: 10.2246 - acc:
 0.3120Epoch 00002: val_loss improved from 10.62197 to 10.36610, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.2288 - acc: 0.312
0 - val_loss: 10.3661 - val_acc: 0.2850
Epoch 4/20
6620/6680 [============================>.] - ETA: 0s - loss: 10.0574 - acc:
 0.3411Epoch 00003: val_loss improved from 10.36610 to 10.27910, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.0543 - acc: 0.341
5 - val_loss: 10.2791 - val_acc: 0.3138
Epoch 5/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.9970 - acc:
 0.3528Epoch 00004: val_loss improved from 10.27910 to 10.24878, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.9755 - acc: 0.3543
- val_loss: 10.2488 - val_acc: 0.3126
Epoch 6/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.8929 - acc:
 0.3666Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.8848 - acc: 0.3671
- val_loss: 10.2616 - val_acc: 0.3090
Epoch 7/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.8137 - acc:
 0.3750Epoch 00006: val_loss improved from 10.24878 to 10.19725, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.8082 - acc: 0.3751
- val_loss: 10.1973 - val_acc: 0.3138
Epoch 8/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.6755 - acc:
 0.3847Epoch 00007: val_loss improved from 10.19725 to 10.08032, saving mode
l to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.6941 - acc: 0.3837
- val_loss: 10.0803 - val_acc: 0.3198
Epoch 9/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.6359 - acc:
 0.3919Epoch 00008: val_loss improved from 10.08032 to 10.03602, saving mode
l to saved_models/weights.best.VGG16.hdf5
```

```
6680/6680 [==============================] - 1s - loss: 9.6324 - acc: 0.3922
- val_loss: 10.0360 - val_acc: 0.3317
Epoch 10/20
6580/6680 [============================>.] - ETA: 0s - loss: 9.5900 - acc:
 0.3960Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.5949 - acc: 0.3955
- val_loss: 10.0388 - val_acc: 0.3353
Epoch 11/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.5181 - acc:
 0.4000Epoch 00010: val_loss improved from 10.03602 to 9.91955, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5010 - acc: 0.4010
- val_loss: 9.9195 - val_acc: 0.3377
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.3142 - acc:
 0.4065Epoch 00011: val_loss improved from 9.91955 to 9.66307, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.3251 - acc: 0.4058
- val_loss: 9.6631 - val_acc: 0.3413
Epoch 13/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.1676 - acc:
 0.4195Epoch 00012: val_loss improved from 9.66307 to 9.61508, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1683 - acc: 0.4196
- val_loss: 9.6151 - val_acc: 0.3617
Epoch 14/20
6640/6680 [============================>.] - ETA: 0s - loss: 9.0416 - acc:
 0.4258Epoch 00013: val_loss improved from 9.61508 to 9.43330, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.0433 - acc: 0.4256
- val_loss: 9.4333 - val_acc: 0.3725
Epoch 15/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.9578 - acc:
 0.4356Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.9706 - acc: 0.4349
- val_loss: 9.4582 - val_acc: 0.3581
Epoch 16/20
6580/6680 [============================>.] - ETA: 0s - loss: 8.7998 - acc:
 0.4384Epoch 00015: val_loss improved from 9.43330 to 9.33256, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7878 - acc: 0.4385
- val_loss: 9.3326 - val_acc: 0.3641
Epoch 17/20
6580/6680 [============================>.] - ETA: 0s - loss: 8.6091 - acc:
 0.4532Epoch 00016: val_loss improved from 9.33256 to 9.17785, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5988 - acc: 0.4539
- val_loss: 9.1779 - val_acc: 0.3820
Epoch 18/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.5702 - acc:
 0.4615Epoch 00017: val_loss improved from 9.17785 to 9.13352, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5575 - acc: 0.4621
- val_loss: 9.1335 - val_acc: 0.3820
Epoch 19/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.5405 - acc:
 0.4661Epoch 00018: val_loss improved from 9.13352 to 9.12029, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5362 - acc: 0.4665
- val_loss: 9.1203 - val_acc: 0.3856
Epoch 20/20
6540/6680 [============================>.] - ETA: 0s - loss: 8.4977 - acc:
 0.4682Epoch 00019: val_loss improved from 9.12029 to 9.08648, saving model
 to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5103 - acc: 0.4675
- val_loss: 9.0865 - val_acc: 0.3856
```

Out[21]:

```
<keras.callbacks.History at 0x7f2dd00f4400>
```

Load the Model with the Best Validation Loss

In [22]:

```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

Test the Model

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

In [23]:

```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axis=0))) for fe

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

```
Test accuracy: 38.7560%
```

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

(IMPLEMENTATION) Model Architecture

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

```
    <your model's name>.summary()
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer:

- More refined approach would be to leverage a network pre-trained on a large dataset. Such a network would have already learned features that are useful for most computer vision problems, and leveraging such features would allow us to reach a better accuracy than any method that would only rely on the

available data.
- We will use the Resnet50 architecture, pre-trained on the ImageNet dataset. Because the ImageNet dataset contains several "Dog" classes (Brittany Dog, American water spaniel Dog...) and many other classes among its total of 1000 classes, this model will already have learned features that are relevant to our classification problem.
- The strategy will be as follow: we will only instantiate the convolutional part of the model, everything up to the fully-connected layers. We will then run this model on our training and validation data once, recording the output (the "bottleneck features" from the Resnet50 model: the last activation maps before the fully-connected layers) in 'bottleneck_features/DogResnet50Data.npz'. Then we will train a small fully-connected model on top of the stored features

Top small fully connected model architecture: We need to flatten the weights before passing to fully connected dense layer for image classification.

- The first layer is flatten layer with one dimensional tensor(vector)
- The second layer is dense(fully connected layer) with 256 neurons with activation function Relu
- We can apply dropout to prevent overfitting. Most of the time dropout is applied to fully connected layers compared to convolutional layers. since the convolutional layers don’t have a lot of parameters, overfitting is not a problem and therefore dropout would not have much effect. However, the additional gain in performance obtained by adding dropout in the convolutional layers (3.02% to 2.55%) is worth noting. Dropout in the lower layers still helps because it provides noisy inputs for the higher fully connected layers which prevents them from overfitting
- The third layer is dense layer with 133 neurons with activation to classify given dog image to probability estimations of the 133 classes.

In [26]:

```
### TODO: Define your architecture.
Resnet50_model = Sequential()
Resnet50_model.add(Flatten(input_shape=train_Resnet50.shape[1:]))
Resnet50_model.add(Dense(256, activation='relu'))
Resnet50_model.add(Dropout(0.5))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_3 (Flatten)          (None, 2048)              0         
_________________________________________________________________
dense_4 (Dense)              (None, 256)               524544    
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 133)               34181     
=================================================================
Total params: 558,725
Trainable params: 558,725
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [27]:

```
### TODO: Compile the model.
Resnet50_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accu
```

(IMPLEMENTATION) Train the Model

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

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

In [28]:

```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.Resnet50.hdf5',
                               verbose=1, save_best_only=True)

Resnet50_model.fit(train_Resnet50, train_targets,
          validation_data=(valid_Resnet50, valid_targets),
          epochs=30, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/30
6600/6680 [============================>.] - ETA: 0s - loss: 3.1367 - acc:
 0.2944Epoch 00000: val_loss improved from inf to 1.16495, saving model to s
aved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 2s - loss: 3.1269 - acc: 0.2952
- val_loss: 1.1649 - val_acc: 0.6958
Epoch 2/30
6500/6680 [============================>.] - ETA: 0s - loss: 1.4173 - acc:
 0.5920Epoch 00001: val_loss improved from 1.16495 to 0.81961, saving model
 to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 1.4138 - acc: 0.5936
- val_loss: 0.8196 - val_acc: 0.7437
Epoch 3/30
6580/6680 [============================>.] - ETA: 0s - loss: 1.0250 - acc:
 0.6986Epoch 00002: val_loss improved from 0.81961 to 0.72856, saving model
 to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 1.0216 - acc: 0.6999
- val_loss: 0.7286 - val_acc: 0.7617
Epoch 4/30
6520/6680 [============================>.] - ETA: 0s - loss: 0.8762 - acc:
 0.7344Epoch 00003: val_loss improved from 0.72856 to 0.63233, saving model
 to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.8782 - acc: 0.7346
- val_loss: 0.6323 - val_acc: 0.8024
Epoch 5/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.7601 - acc:
 0.7679Epoch 00004: val_loss improved from 0.63233 to 0.61164, saving model
 to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.7593 - acc: 0.7684
- val_loss: 0.6116 - val_acc: 0.8132
Epoch 6/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.6427 - acc:
 0.7948Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.6425 - acc: 0.7954
- val_loss: 0.6358 - val_acc: 0.7952
Epoch 7/30
6520/6680 [============================>.] - ETA: 0s - loss: 0.5870 - acc:
 0.8166Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.5874 - acc: 0.8172
- val_loss: 0.6418 - val_acc: 0.8084
Epoch 8/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.5530 - acc:
 0.8229Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.5509 - acc: 0.8231
- val_loss: 0.6928 - val_acc: 0.8096
Epoch 9/30
6540/6680 [============================>.] - ETA: 0s - loss: 0.5170 - acc:
 0.8384Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.5178 - acc: 0.8380
- val_loss: 0.6254 - val_acc: 0.8156
```

```
Epoch 10/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.4713 - acc:
 0.8535Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.4724 - acc: 0.8531
- val_loss: 0.6199 - val_acc: 0.8299
Epoch 11/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.4380 - acc:
 0.8612Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.4398 - acc: 0.8603
- val_loss: 0.6154 - val_acc: 0.8108
Epoch 12/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.4351 - acc:
 0.8662Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.4343 - acc: 0.8668
- val_loss: 0.6752 - val_acc: 0.8048
Epoch 13/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.3943 - acc:
 0.8773Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3918 - acc: 0.8780
- val_loss: 0.7140 - val_acc: 0.8240
Epoch 14/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.3834 - acc:
 0.8813Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3839 - acc: 0.8811
- val_loss: 0.7159 - val_acc: 0.8168
Epoch 15/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.3758 - acc:
 0.8809Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3775 - acc: 0.8805
- val_loss: 0.7348 - val_acc: 0.8204
Epoch 16/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.3557 - acc:
 0.8921Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3591 - acc: 0.8916
- val_loss: 0.6495 - val_acc: 0.8347
Epoch 17/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.3405 - acc:
 0.8947Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3408 - acc: 0.8948
- val_loss: 0.7728 - val_acc: 0.8216
Epoch 18/30
6620/6680 [============================>.] - ETA: 0s - loss: 0.3196 - acc:
 0.8965Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3193 - acc: 0.8964
- val_loss: 0.7918 - val_acc: 0.8299
Epoch 19/30
6620/6680 [============================>.] - ETA: 0s - loss: 0.3490 - acc:
 0.8977Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3473 - acc: 0.8981
- val_loss: 0.7708 - val_acc: 0.8347
Epoch 20/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.3074 - acc:
 0.9078Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3086 - acc: 0.9075
- val_loss: 0.7033 - val_acc: 0.8347
Epoch 21/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.3022 - acc:
 0.9099Epoch 00020: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3008 - acc: 0.9103
- val_loss: 0.7232 - val_acc: 0.8240
Epoch 22/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.3020 - acc:
 0.9106Epoch 00021: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3011 - acc: 0.9106
- val_loss: 0.8299 - val_acc: 0.8323
Epoch 23/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.2987 - acc:
 0.9140Epoch 00022: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2970 - acc: 0.9142
- val_loss: 0.7995 - val_acc: 0.8347
Epoch 24/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.3061 - acc:
 0.9097Epoch 00023: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.3095 - acc: 0.9096
- val_loss: 0.6651 - val_acc: 0.8431
Epoch 25/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.2906 - acc:
 0.9149Epoch 00024: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2909 - acc: 0.9147
- val_loss: 0.9177 - val_acc: 0.8263
Epoch 26/30
6520/6680 [============================>.] - ETA: 0s - loss: 0.2896 - acc:
 0.9175Epoch 00025: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2917 - acc: 0.9166
- val_loss: 0.7894 - val_acc: 0.8407
Epoch 27/30
6540/6680 [============================>.] - ETA: 0s - loss: 0.2629 - acc:
 0.9213Epoch 00026: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2646 - acc: 0.9211
- val_loss: 0.8948 - val_acc: 0.8299
Epoch 28/30
6600/6680 [============================>.] - ETA: 0s - loss: 0.2785 - acc:
 0.9214Epoch 00027: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2772 - acc: 0.9214
- val_loss: 0.8684 - val_acc: 0.8347
Epoch 29/30
6620/6680 [============================>.] - ETA: 0s - loss: 0.2570 - acc:
 0.9254Epoch 00028: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2577 - acc: 0.9256
- val_loss: 0.8278 - val_acc: 0.8419
Epoch 30/30
6580/6680 [============================>.] - ETA: 0s - loss: 0.2602 - acc:
 0.9289Epoch 00029: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 0.2608 - acc: 0.9283
- val_loss: 0.9150 - val_acc: 0.8263
```

Out[28]:

```
<keras.callbacks.History at 0x7f2db19dbfd0>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [29]:

```
### TODO: Load the model weights with the best validation loss.
Resnet50_model.load_weights('saved_models/weights.best.Resnet50.hdf5')
```

(IMPLEMENTATION) Test the Model

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

In [30]:

```
### TODO: Calculate classification accuracy on the test dataset.
Resnet50_predictions = [np.argmax(Resnet50_model.predict(np.expand_dims(feature, axis=0)))

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

```
Test accuracy: 79.6651%
```

(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.
from extract_bottleneck_features import *

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

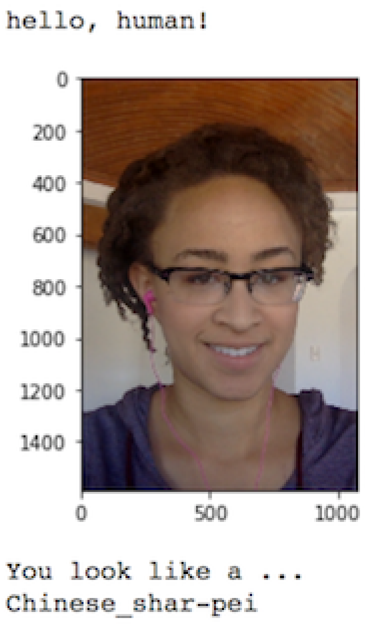
Step 6: Write your Algorithm

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

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

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

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



(IMPLEMENTATION) Write your Algorithm

In [32]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def predict_dog_breed(img_path):
    if(dog_detector(img_path)):
        return Resnet50_predict_breed(img_path)
    elif(face_detector(img_path)):
        return Resnet50_predict_breed(img_path)
    else:
        return 'an error'
```

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 as expected. The algorithm predicted correctly for the given dog images.

- To further improve our previous result, we can try to "fine-tune" the last convolutional block of the Resnet50 model alongside the top-level classifier
- we choose to only fine-tune the last convolutional block rather than the entire network in order to prevent overfitting, since the entire network would have a very large entropic capacity and thus a strong tendency to overfit. The features learned by low-level convolutional blocks are more general, less abstract than those found higher-up, so it is sensible to keep the first few blocks fixed (more general features) and only fine-tune the last one (more specialized features).
- fine-tuning should be done with a very slow learning rate, and typically with the SGD optimizer rather than an adaptative learning rate optimizer such as RMSProp. This is to make sure that the magnitude of the updates stays very small, so as not to wreck the previously learned features.

In [39]:

```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
test_imgs = np.array(glob("test_images/*"))
for img in test_imgs:
    print(predict_dog_breed(img))
```

```
French_bulldog
Akita
Labrador_retriever
Australian_cattle_dog
Chinese_crested
Boxer
```

In [ ]:

```
```