

Artificial Intelligence Nanodegree

Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

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

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

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

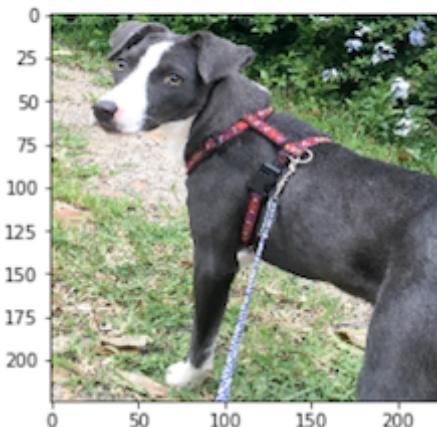
Note: Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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

Why We're Here

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

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



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

The Road Ahead

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

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

Step 0: Import Datasets

Import Dog Dataset

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

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

In [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_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_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)

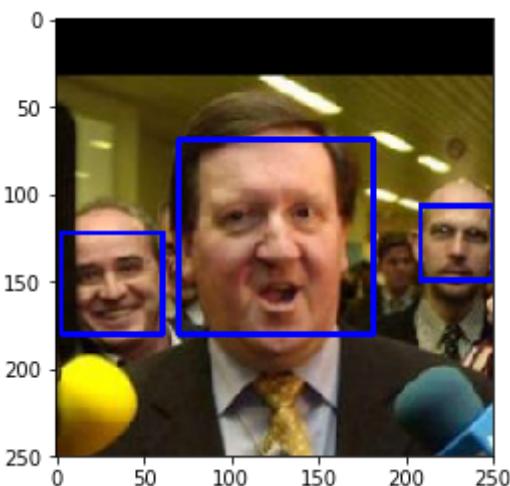
# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

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

/home/aind2/anaconda3/envs/aind2/lib/python3.6/site-packages/matplotlib/font_manager.py:280: UserWarning: Matplotlib is building the font cache using fc-list. This may take a moment.

'Matplotlib is building the font cache using fc-list.'

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.

def percentage_of_faces(files):
    counter = 0

    for f in files:
        if face_detector(f):
            counter += 1

    return counter/len(files)

per_dog = float(percentage_of_faces(dog_files_short)*100.00)
per_human = float(percentage_of_faces(human_files_short)*100.00)

print('The percentage of Dog faces detected as human face is % .2f%%' % per_dog)
print('The percentage of Human faces detected as human face is % .2f%%' % per_human)

```

The percentage of Dog faces detected as human face is 11.00%
The percentage of Human faces detected as human face is 99.00%

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 unnecessarily 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:

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-mode  
ls/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kerne  
l.s.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

(nb_samples, rows, columns, 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

(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/fc15e7c15f6e8d7363c41854d24d77f6) (<https://gist.github.com/yrevar/fc15e7c15f6e8d7363c41854d24d77f6>), 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.

def percentage_of_dogs(files):
    counter = 0

    for f in files:
        if dog_detector(f):
            counter += 1

    return counter/len(files)

per_dog = float(percentage_of_dogs(dog_files_short)*100.00)
per_human = float(percentage_of_dogs(human_files_short)*100.00)

print('The percentage of Dog faces detected as dog is % 3.2f%%' % per_dog)
print('The percentage of Human faces detected as dog is % 3.2f%%' % per_human)
```

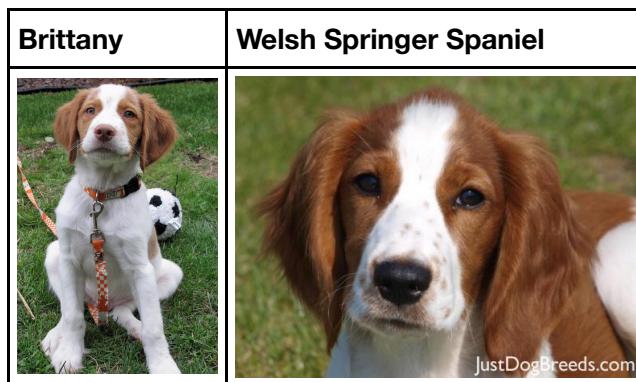
The percentage of Dog faces detected as dog is 100.00%
The percentage of Human faces detected as dog is 0.00%

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

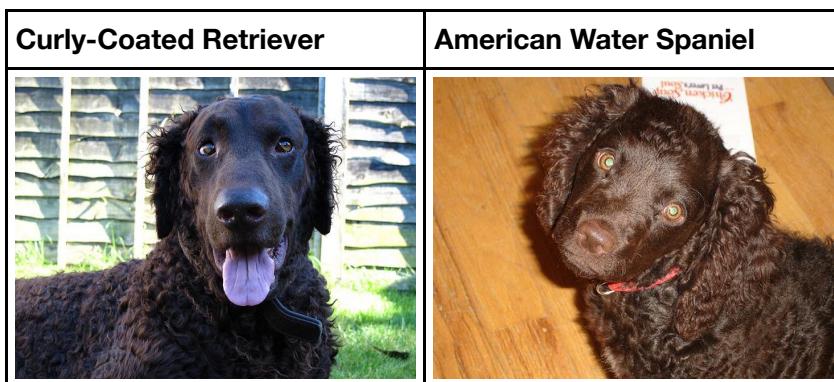
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning yet!), and you must attain a test accuracy of at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



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



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 imbalanced, 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]:

```
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, 125.19it/s]
 100%|██████████| 835/835 [00:05<00:00, 140.85it/s]
 100%|██████████| 836/836 [00:05<00:00, 141.25it/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	INPUT
max_pooling2d_1 (MaxPooling2D)	(None, 111, 111, 16)	0	CONV
conv2d_2 (Conv2D)	(None, 110, 110, 32)	2080	POOL
max_pooling2d_2 (MaxPooling2D)	(None, 55, 55, 32)	0	CONV
conv2d_3 (Conv2D)	(None, 54, 54, 64)	8256	POOL
max_pooling2d_3 (MaxPooling2D)	(None, 27, 27, 64)	0	CONV
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 64)	0	GAP
dense_1 (Dense)	(None, 133)	8645	DENSE
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:

In [12]:

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

model = Sequential()

# The next architecture is based on the CIFAR10 Winners architecture
# http://blog.kaggle.com/2015/01/02/cifar-10-competition-winners-interviews-with
-dr-ben-graham-phil-culliton-zygmunt-zajac/
# https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py
# The idea was to have a base from a known architecture. In the Keras example they removed a couple of layers
# (I assume to avoid overfitting), however I also follow the structure defined in the article and the one proposed
# for the second prize (specially the dropouts values). This dropout theory is quiet impressive and I think that
# I can mix both architectures and having a good performance
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu',
                 input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.083333)) # Phil Culliton DropOut

model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.208333)) # Phil Culliton DropOut

# Second Experiment (new layers)

model.add(Conv2D(filters=96, kernel_size=2, padding='same', activation='relu'))
model.add(Conv2D(filters=96, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3)) # Phil Culliton DropOut

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(133))

model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 224, 224, 32)	416
max_pooling2d_2 (MaxPooling2D)	(None, 112, 112, 32)	0
dropout_1 (Dropout)	(None, 112, 112, 32)	0
conv2d_2 (Conv2D)	(None, 112, 112, 64)	8256
conv2d_3 (Conv2D)	(None, 112, 112, 64)	16448
max_pooling2d_3 (MaxPooling2D)	(None, 56, 56, 64)	0
dropout_2 (Dropout)	(None, 56, 56, 64)	0
conv2d_4 (Conv2D)	(None, 56, 56, 96)	24672
conv2d_5 (Conv2D)	(None, 56, 56, 96)	36960
max_pooling2d_4 (MaxPooling2D)	(None, 28, 28, 96)	0
dropout_3 (Dropout)	(None, 28, 28, 96)	0
flatten_2 (Flatten)	(None, 75264)	0
dense_1 (Dense)	(None, 512)	38535680
activation_50 (Activation)	(None, 512)	0
dropout_4 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 133)	68229

Total params: 38,690,661
Trainable params: 38,690,661
Non-trainable params: 0

Compile the Model

In [13]:

```
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) (<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>), but this is not a requirement.

In [22]:

```
from keras.callbacks import ModelCheckpoint
from keras.preprocessing.image import ImageDataGenerator

### TODO: specify the number of epochs that you would like to use to train the model.
data_augmentation = True
batch_size = 20
epochs = 5

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

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

if not data_augmentation:
    model.fit(train_tensors, train_targets,
              validation_data=(valid_tensors, valid_targets),
              epochs=epochs, batch_size=batch_size, callbacks=[checkpointer], verbose=1)

else:
    datagen_train = ImageDataGenerator(width_shift_range=0.1,
                                        height_shift_range=0.1, horizontal_flip=True)
    datagen_valid = ImageDataGenerator(width_shift_range=0.1,
                                        height_shift_range=0.1, horizontal_flip=True)
    model.fit_generator(datagen_train.flow(train_tensors, train_targets, batch_size=batch_size),
                        steps_per_epoch=train_tensors.shape[0] // batch_size,
                        epochs=epochs, callbacks=[checkpointer], verbose=1,
                        validation_data=datagen_valid.flow(valid_tensors, valid_targets, batch_size=batch_size),
                        validation_steps=valid_tensors.shape[0] // batch_size)
```

```

Epoch 1/5
333/334 [=====>.] - ETA: 2s - loss: 8.4021 -
acc: 0.0080Epoch 00000: val_loss improved from inf to 8.16047, saving model to saved_models/weights.best.from_scratch.hdf5
334/334 [=====] - 1033s - loss: 8.4055 - acc: 0.0079 - val_loss: 8.1605 - val_acc: 0.0073
Epoch 2/5
333/334 [=====>.] - ETA: 2s - loss: 8.4033 -
acc: 0.0093Epoch 00001: val_loss improved from 8.16047 to 8.08096, saving model to saved_models/weights.best.from_scratch.hdf5
334/334 [=====] - 1032s - loss: 8.4063 - acc: 0.0093 - val_loss: 8.0810 - val_acc: 0.0110
Epoch 3/5
333/334 [=====>.] - ETA: 2s - loss: 8.4366 -
acc: 0.0090Epoch 00002: val_loss improved from 8.08096 to 7.93050, saving model to saved_models/weights.best.from_scratch.hdf5
334/334 [=====] - 1033s - loss: 8.4505 - acc: 0.0090 - val_loss: 7.9305 - val_acc: 0.0123
Epoch 4/5
333/334 [=====>.] - ETA: 2s - loss: 8.2599 -
acc: 0.0102Epoch 00003: val_loss did not improve
334/334 [=====] - 1001s - loss: 8.2599 - acc: 0.0102 - val_loss: 8.5448 - val_acc: 0.0123
Epoch 5/5
333/334 [=====>.] - ETA: 2s - loss: 8.1942 -
acc: 0.0102Epoch 00004: val_loss did not improve
334/334 [=====] - 1001s - loss: 8.2058 - acc: 0.0102 - val_loss: 8.0492 - val_acc: 0.0086

```

Load the Model with the Best Validation Loss

In [24]:

```
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 [25]:

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

Test accuracy: 1.5550%

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 [26]:

```
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 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 [27]:

```
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 #
<hr/>		
global_average_pooling2d_1 ((None, 512)	0
dense_3 (Dense)	(None, 133)	68229
<hr/>		
Total params: 68,229		
Trainable params: 68,229		
Non-trainable params: 0		

Compile the Model

In [28]:

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

Train the Model

In [29]:

```
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
6500/6680 [=====>..] - ETA: 0s - loss: 12.4160
- acc: 0.1191Epoch 00000: val_loss improved from inf to 11.03585, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 12.4054 - acc: 0.1198 - val_loss: 11.0359 - val_acc: 0.2108
Epoch 2/20
6440/6680 [=====>..] - ETA: 0s - loss: 10.5189
- acc: 0.2593Epoch 00001: val_loss improved from 11.03585 to 10.48331, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 10.5172 - acc: 0.2608 - val_loss: 10.4833 - val_acc: 0.2611
Epoch 3/20
6440/6680 [=====>..] - ETA: 0s - loss: 9.9744
- acc: 0.3214 Epoch 00002: val_loss improved from 10.48331 to 10.23157, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.9556 - acc: 0.3225 - val_loss: 10.2316 - val_acc: 0.2946
Epoch 4/20
6440/6680 [=====>..] - ETA: 0s - loss: 9.7681
- acc: 0.3539Epoch 00003: val_loss improved from 10.23157 to 10.17218, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.7637 - acc: 0.3539 - val_loss: 10.1722 - val_acc: 0.3006
Epoch 5/20
6420/6680 [=====>..] - ETA: 0s - loss: 9.6529
- acc: 0.3726Epoch 00004: val_loss improved from 10.17218 to 10.12301, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.6569 - acc: 0.3725 - val_loss: 10.1230 - val_acc: 0.3078
Epoch 6/20
6400/6680 [=====>..] - ETA: 0s - loss: 9.5525
- acc: 0.3833Epoch 00005: val_loss improved from 10.12301 to 9.88702, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.5338 - acc: 0.3850 - val_loss: 9.8870 - val_acc: 0.3257
Epoch 7/20
6440/6680 [=====>..] - ETA: 0s - loss: 9.3861
- acc: 0.4002Epoch 00006: val_loss did not improve
6680/6680 [=====] - 1s - loss: 9.3990 - acc: 0.3994 - val_loss: 9.9167 - val_acc: 0.3353
Epoch 8/20
6440/6680 [=====>..] - ETA: 0s - loss: 9.3697
- acc: 0.4053Epoch 00007: val_loss improved from 9.88702 to 9.79356, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.3591 - acc: 0.4060 - val_loss: 9.7936 - val_acc: 0.3353
Epoch 9/20
6440/6680 [=====>..] - ETA: 0s - loss: 9.2357
- acc: 0.4110Epoch 00008: val_loss improved from 9.79356 to 9.72521, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.2500 - acc: 0.4103 - val_loss: 9.7252 - val_acc: 0.3449
Epoch 10/20
6480/6680 [=====>..] - ETA: 0s - loss: 9.2028
- acc: 0.4167Epoch 00009: val_loss improved from 9.72521 to 9.64050, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.1962 - acc: 0.4169 - val_loss: 9.6405 - val_acc: 0.3473
Epoch 11/20

```
6420/6680 [=====>..] - ETA: 0s - loss: 9.0707
- acc: 0.4241Epoch 00010: val_loss improved from 9.64050 to 9.5835
9, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 9.0995 - ac
c: 0.4222 - val_loss: 9.5836 - val_acc: 0.3461
Epoch 12/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.9595
- acc: 0.4337Epoch 00011: val_loss improved from 9.58359 to 9.4625
4, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.9503 - ac
c: 0.4343 - val_loss: 9.4625 - val_acc: 0.3605
Epoch 13/20
6500/6680 [=====>..] - ETA: 0s - loss: 8.8939
- acc: 0.4428Epoch 00012: val_loss did not improve
6680/6680 [=====] - 1s - loss: 8.9119 - ac
c: 0.4410 - val_loss: 9.5461 - val_acc: 0.3437
Epoch 14/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.8410
- acc: 0.4427Epoch 00013: val_loss improved from 9.46254 to 9.3872
9, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.8234 - ac
c: 0.4440 - val_loss: 9.3873 - val_acc: 0.3713
Epoch 15/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.7844
- acc: 0.4505Epoch 00014: val_loss improved from 9.38729 to 9.3801
8, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.7906 - ac
c: 0.4500 - val_loss: 9.3802 - val_acc: 0.3725
Epoch 16/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.6206
- acc: 0.4472Epoch 00015: val_loss improved from 9.38018 to 9.1617
7, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.6282 - ac
c: 0.4463 - val_loss: 9.1618 - val_acc: 0.3665
Epoch 17/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.4066
- acc: 0.4658Epoch 00016: val_loss improved from 9.16177 to 9.0346
0, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.3954 - ac
c: 0.4666 - val_loss: 9.0346 - val_acc: 0.3796
Epoch 18/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.3508
- acc: 0.4728Epoch 00017: val_loss improved from 9.03460 to 8.9778
6, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.3399 - ac
c: 0.4735 - val_loss: 8.9779 - val_acc: 0.3832
Epoch 19/20
6420/6680 [=====>..] - ETA: 0s - loss: 8.2979
- acc: 0.4768Epoch 00018: val_loss improved from 8.97786 to 8.9462
2, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.2909 - ac
c: 0.4771 - val_loss: 8.9462 - val_acc: 0.3904
Epoch 20/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.2465
- acc: 0.4807Epoch 00019: val_loss did not improve
6680/6680 [=====] - 1s - loss: 8.2619 - ac
c: 0.4799 - val_loss: 8.9586 - val_acc: 0.3904
```

Out[29]:

<keras.callbacks.History at 0x7fc64a2f8dd8>

Load the Model with the Best Validation Loss

In [30]:

```
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 [31]:

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

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

Test accuracy: 39.2344%

Predict Dog Breed with the Model

In [32]:

```
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 \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz) bottleneck features
- [ResNet-50 \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz) bottleneck features
- [Inception \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz\)](https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz) bottleneck features
- [Xception \(https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogXceptionData.npz\)](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 [33]:

```
### 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:

In [43]:

```
### TODO: Define your architecture.
Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[1:]))

Resnet50_model.add(Dropout(0.3)) # Phil Culliton DropOut

# Last layers of my original
# Resnet50_model.add(Dense(512)) This layer was generating a lot of error
Resnet50_model.add(Activation('relu'))
Resnet50_model.add(Dropout(0.5)) #Final Dropout
Resnet50_model.add(Dense(133, activation='softmax')) # After I added this layer
# with an activation of softmax the result improved almost a 100%
#Resnet50_model.add(Dense(133)) This layer wasn't working

# I think is suitable because it's mixing all the knowledge from Resnet50 and the
# studies of the professor Culliton (about Dropouts)
# The idea of this last steps it's to reduce the overfitting at most as it's possible
# The scratch experiment show us that the performance of my net was really good,
# so I decided to reuse it.
# However I made a lot of changes like the use of softmax for the last activation
# function and the removal of some layers

Resnet50_model.summary()
```

Layer (type)	Output Shape	Param #
<hr/>		
global_average_pooling2d_5 ((None, 2048)	0
dropout_10 (Dropout)	(None, 2048)	0
activation_53 (Activation)	(None, 2048)	0
dropout_11 (Dropout)	(None, 2048)	0
dense_7 (Dense)	(None, 133)	272517
<hr/>		
Total params:	272,517	
Trainable params:	272,517	
Non-trainable params:	0	

(IMPLEMENTATION) Compile the Model

In [44]:

```
### TODO: Compile the model.  
Resnet50_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', 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 [49]:

```
### 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=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

Train on 6680 samples, validate on 835 samples
Epoch 1/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3713
- acc: 0.9101Epoch 00000: val_loss improved from inf to 0.80828, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [=====] - 1s - loss: 0.3732 - acc: 0.9100 - val_loss: 0.8083 - val_acc: 0.8311
Epoch 2/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3635
- acc: 0.9154Epoch 00001: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3633 - acc: 0.9153 - val_loss: 0.8564 - val_acc: 0.8192
Epoch 3/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3764
- acc: 0.9160Epoch 00002: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3748 - acc: 0.9162 - val_loss: 0.8643 - val_acc: 0.8204
Epoch 4/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3681
- acc: 0.9145Epoch 00003: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3659 - acc: 0.9150 - val_loss: 0.8589 - val_acc: 0.8216
Epoch 5/20
6660/6680 [=====>.] - ETA: 0s - loss: 0.3765
- acc: 0.9116Epoch 00004: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3776 - acc: 0.9117 - val_loss: 0.8623 - val_acc: 0.8287
Epoch 6/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3542
- acc: 0.9181Epoch 00005: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3546 - acc: 0.9178 - val_loss: 0.8497 - val_acc: 0.8240
Epoch 7/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3463
- acc: 0.9190Epoch 00006: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3457 - acc: 0.9190 - val_loss: 0.8563 - val_acc: 0.8144
Epoch 8/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3654
- acc: 0.9186Epoch 00007: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3663 - acc: 0.9184 - val_loss: 0.8411 - val_acc: 0.8347
Epoch 9/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3464
- acc: 0.9181Epoch 00008: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3466 - acc: 0.9178 - val_loss: 0.8742 - val_acc: 0.8287
Epoch 10/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3533
- acc: 0.9177Epoch 00009: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3534 - acc: 0.9175 - val_loss: 0.8483 - val_acc: 0.8228
Epoch 11/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3535
- acc: 0.9172Epoch 00010: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3536 - acc: 0.9169 - val_loss: 0.8658 - val_acc: 0.8347
Epoch 12/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3721
- acc: 0.9157Epoch 00011: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3692 - acc:

```
c: 0.9165 - val_loss: 0.8785 - val_acc: 0.8299
Epoch 13/20
6660/6680 [=====>.] - ETA: 0s - loss: 0.3395
- acc: 0.9231Epoch 00012: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3393 - ac
c: 0.9231 - val_loss: 0.8681 - val_acc: 0.8263
Epoch 14/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3575
- acc: 0.9222Epoch 00013: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3562 - ac
c: 0.9226 - val_loss: 0.8691 - val_acc: 0.8395
Epoch 15/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3449
- acc: 0.9246Epoch 00014: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3440 - ac
c: 0.9247 - val_loss: 0.8470 - val_acc: 0.8311
Epoch 16/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3317
- acc: 0.9220Epoch 00015: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3303 - ac
c: 0.9222 - val_loss: 0.8561 - val_acc: 0.8347
Epoch 17/20
6640/6680 [=====>.] - ETA: 0s - loss: 0.3590
- acc: 0.9217Epoch 00016: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3579 - ac
c: 0.9219 - val_loss: 0.8818 - val_acc: 0.8311
Epoch 18/20
6460/6680 [=====>.] - ETA: 0s - loss: 0.3426
- acc: 0.9198Epoch 00017: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3440 - ac
c: 0.9195 - val_loss: 0.8653 - val_acc: 0.8228
Epoch 19/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3262
- acc: 0.9255Epoch 00018: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3262 - ac
c: 0.9257 - val_loss: 0.9223 - val_acc: 0.8192
Epoch 20/20
6620/6680 [=====>.] - ETA: 0s - loss: 0.3493
- acc: 0.9231Epoch 00019: val_loss did not improve
6680/6680 [=====] - 1s - loss: 0.3474 - ac
c: 0.9234 - val_loss: 0.9102 - val_acc: 0.8299
```

Out[49]:

<keras.callbacks.History at 0x7fc629a69550>

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [50]:

```
### 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 [51]:

```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
Resnet50_predictions = [np.argmax(Resnet50_model.predict(np.expand_dims(feature,
axis=0))) for feature in test_Resnet50]

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

Test accuracy: 82.7751%

(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 [52]:

```
### 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 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 [58]:

```
### TODO: Write your algorithm.  
### Feel free to use as many code cells as needed.  
  
def show_img(img_path):  
  
    # load color (BGR) image  
    img = cv2.imread(img_path)  
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  
    plt.imshow(cv_rgb)  
    plt.show()  
  
def guess_who(img_path):  
    if dog_detector(img_path):  
        print('Is a Dog!')  
        show_img(img_path)  
        print(Resnet50_predict_breed(img_path))  
  
    elif face_detector(img_path):  
        print('Is a Human!')  
        show_img(img_path)  
        print('It looks like a ' + Resnet50_predict_breed(img_path))  
  
    else:  
        print('Is not a dog and is not human! What could it be? ....')
```

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.

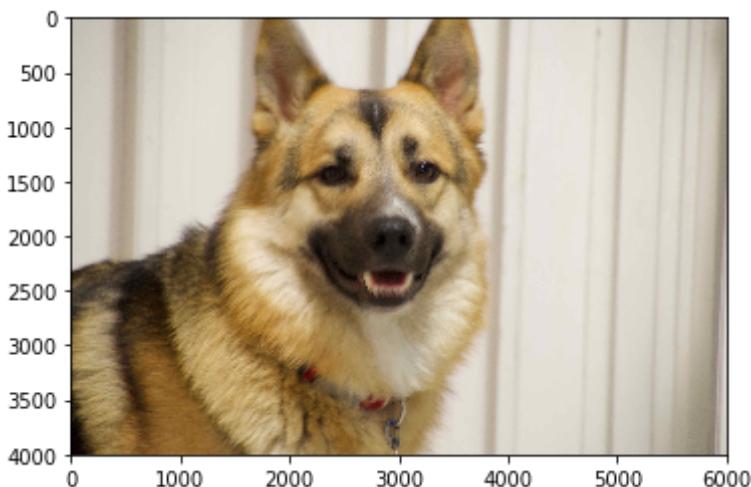
Answer:

In [60]:

```
## 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.
images = ['aind-1.jpg', 'aind-2.jpg', 'aind-3.jpg', 'aind-4.jpg', 'aind-5.jpg',
'aind-6.jpg', 'aind-7.jpg', 'aind-8.jpg']

for img in images:
    guess_who('images/' + img)
    print('\n')
```

Is a Dog!



German_shepherd_dog

Is a Dog!



Belgian_malinois

Is a Dog!



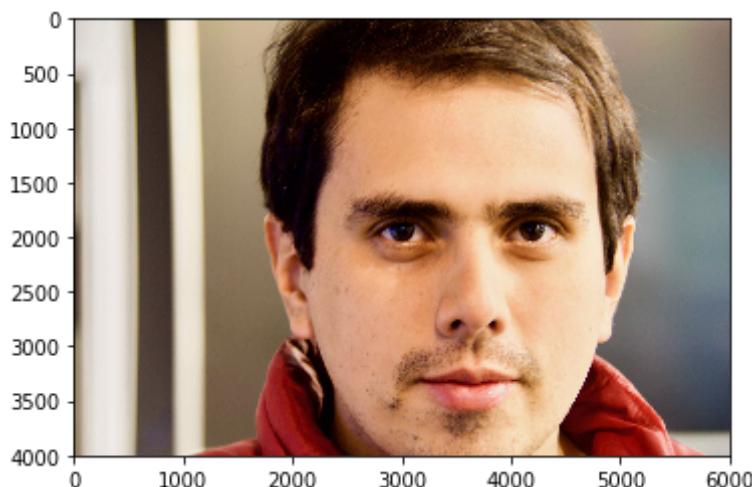
American_eskimo_dog

Is a Dog!



American_eskimo_dog

Is a Human!



It looks like a Beagle

Is a Human!



It looks like a English_springer_spaniel

Is a Dog!



Labrador_retriever

Is a Dog!



Cane_corso

Answer:

Actually I'm quiet impressed, It's obvious that there are some bugs. For instance my first dog is a mix, so I understand that is not easy to detect a specific breed, but it recognized german sheperd (his mom it's a GS). My second dog is a Swiss White sheperd, but that breed wasn't in the list. However the breed that the model showed is quiet similar (at least as a puppy). In my case, now i know that i look like a beagle and my brother as a English Springer spaniel XD.

Possible points of improvement:

- More epochs are needed (I just ran 20)
- Different angles for the dogs, in some cases the false positives are because certain angles are not in the images (even when you use augmentation)
- Use augmentation
- More pictures per breed
- Understand better the filters and to investigate which filters are good to identify dogs