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.

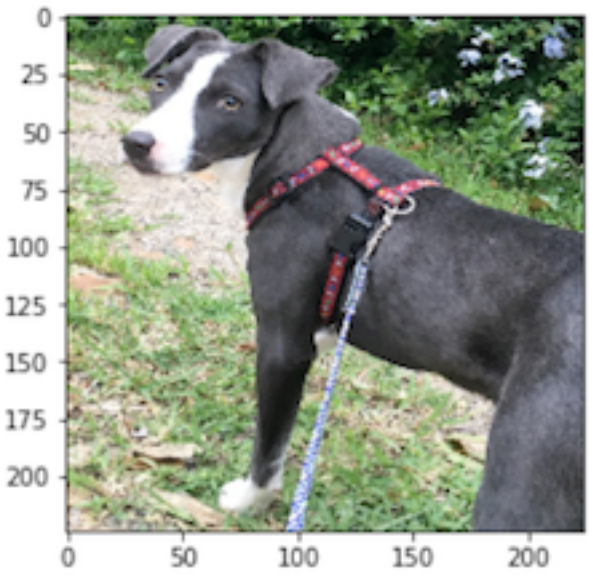
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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 [2]:

```
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('/data/dog_images/train')
valid_files, valid_targets = load_dataset('/data/dog_images/valid')
test_files, test_targets = load_dataset('/data/dog_images/test')

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

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_f
iles, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

```
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 [3]:

```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("/data/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

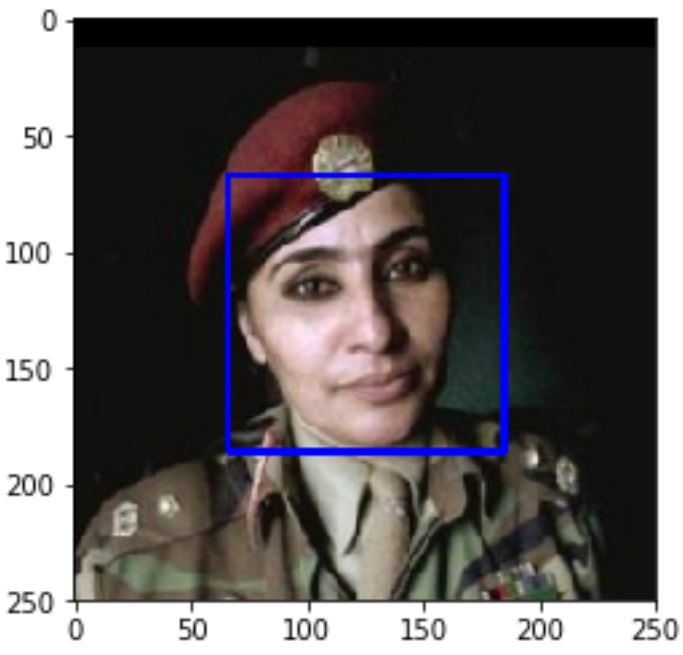
We use OpenCV's implementation of [Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html)](http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on [github (https://github.com/opencv/opencv/tree/master/data/haarcascades)](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 [4]:

```
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: 1
```



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

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

Write a Human Face Detector

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

In [5]:

```
# 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: You can see from the code cell below that the following results were obtained:

- Human faces detected in 100 of the human_files_short 100.00%
- Human faces detected in 11 of the dog_files_short 11.00%

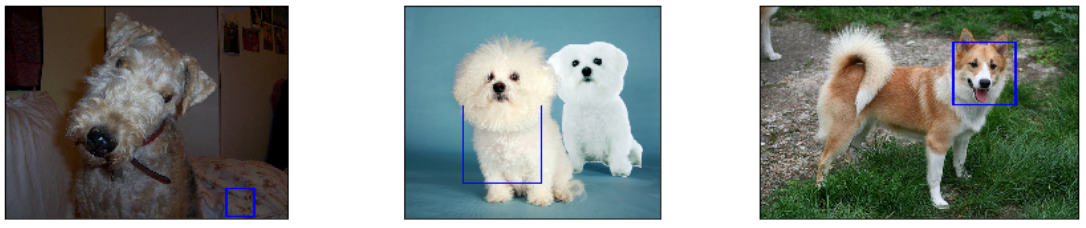
The results show 100% accuracy in detecting at least one face in the first 100 files in the human_faces list. Investigating where the faces were detected in the first 100 dog_files you can see there was actually a human face in one of the pictures. This I believe is accurate and so the true percentage of mis-detected faces would be 10%. Its interesting to see in some of these mis-detections how small the highlighted area where a human face was found can be.

In [6]:

```
def face_detector_display(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    # 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
    return cv_rgb

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.

#check human_files_short
counter = 0
for image in human_files_short:
    counter = counter + face_detector(image)
print ('Human faces detected in',counter, 'of the human_files_short', '{:0.2f}
%'.format((counter/100)*100))
```

```
#check dog_files_short
counter = 0
humans_in_dog_files = []
for image in dog_files_short:
    if face_detector(image):
        humans_in_dog_files.append(image)
        counter = counter + 1
#Lets see if there are any human faces in the dog pictures
print ('Human faces detected in',counter, 'of the dog_files_short', '{:0.2f}%'
.format((counter/100)*100))
print ('Here are the images showing where a human face was detected')
print (humans_in_dog_files)

fig = plt.figure(figsize=(20,10))
for i in range (len(humans_in_dog_files)):
    img = face_detector_display(humans_in_dog_files[i])
    ax = fig.add_subplot(3, 4, i+1, xticks=[], yticks=[])
    ax.imshow(img)
```

Human faces detected in 100 of the human_files_short 100.00%
Human faces detected in 11 of the dog_files_short 11.00%
Here are the images showing where a human face was detected
['/data/dog_images/train/095.Kuvasz/Kuvasz_06442.jpg', '/data/dog_images/train/099.Lhasa_apso/Lhasa_apso_06646.jpg', '/data/dog_images/train/009.American_water_spaniel/American_water_spaniel_00628.jpg', '/data/dog_images/train/057.Dalmatian/Dalmatian_04023.jpg', '/data/dog_images/train/096.Labrador_retriever/Labrador_retriever_06474.jpg', '/data/dog_images/train/106.Newfoundland/Newfoundland_06989.jpg', '/data/dog_images/train/117.Pekingese/Pekingese_07559.jpg', '/data/dog_images/train/039.Bull_terrier/Bull_terrier_02805.jpg', '/data/dog_images/train/097.Lakeland_terrier/Lakeland_terrier_06516.jpg', '/data/dog_images/train/024.Bichon_frise/Bichon_frise_01771.jpg', '/data/dog_images/train/084.Icelandic_sheepdog/Icelandic_sheepdog_05705.jpg']



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: For this particular project I think detecting faces with a clear view is reasonable. If this was placed into an application this could be explicitly stated and would simplify the process and reduce compute overhead. The current method does consider 'scale invariance' i.e. the size of the face, so this would not be an issue, but does not consider other positioning.

For example, there could be a method to detect faces at different profile angles, possibly realigning them prior to detection. Additionaly if a face was in profile where only half the face is visible this again could require a different detection method. These additional complexities could be examined in a future project.

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

In [6]:

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

Step 2: Detect Dogs

In this section, we use a pre-trained [ResNet-50 (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006)](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/)](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)](https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a). Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

In [7]:

```
from keras.applications.resnet50 import ResNet50

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

```
Downloading data from https://github.com/fchollet/deep-learning-mo
dels/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_ke
rnels.h5
102858752/102853048 [==============================] - 1s 0us/step
```

Pre-process the Data

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

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

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

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

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

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

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

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

In [8]:

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

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

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

Making Predictions with ResNet-50

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

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

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

In [9]:

```
from keras.applications.resnet50 import preprocess_input, decode_predictions

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

Write a Dog Detector

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

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

In [10]:

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

(IMPLEMENTATION) Assess the Dog Detector

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

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

Answer: From the results from our code cell below the following results are found.

- Dogs detected in 0 of the human_files_short 0.00%
- Dogs detected in 100 of the dog_files_short 100.00%

When run, our dog_detector finds no dogs in the human_files_short list, and reports 100 files with dogs detected in the dog_files_short list. Unless there is a dog actually in the human_files_short images, the match looks perfect.

In [11]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
counter = 0
for human_file in human_files_short:
    counter = counter + dog_detector(human_file)
print ('Dogs detected in',counter, 'of the human_files_short', '{:0.2f}%'.form
at((counter/100)*100))

counter = 0
for dog_file in dog_files_short:
    counter = counter + dog_detector(dog_file)
print ('Dogs detected in',counter, 'of the dog_files_short', '{:0.2f}%'.format
((counter/100)*100))
```

```
Dogs detected in 0 of the human_files_short 0.00%
Dogs detected in 100 of the dog_files_short 100.00%
```

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

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

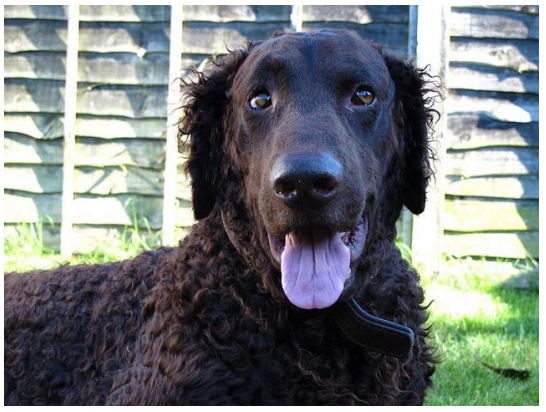
Be careful with adding too many trainable layers! More parameters means longer training, which means you are more likely to need a GPU to accelerate the training process. Thankfully, Keras provides a handy estimate of the time that each epoch is likely to take; you can extrapolate this estimate to figure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

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

Brittany	Welsh Springer Spaniel
	

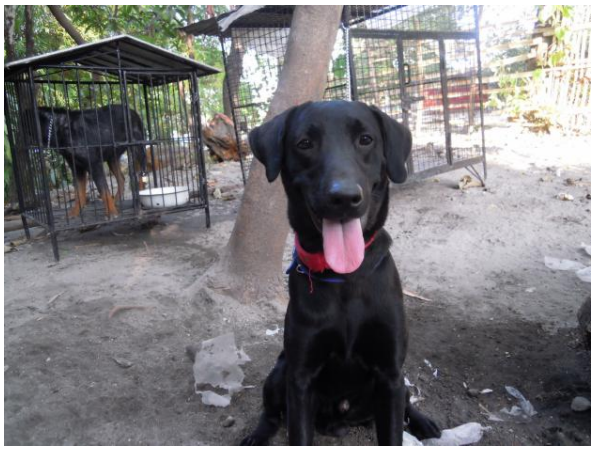
It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever **American Water Spaniel**




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

Yellow Labrador **Chocolate Labrador** **Black Labrador**

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

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

Pre-process the Data

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

In [12]:

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# pre-process the data for Keras
train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|██████████| 6680/6680 [01:27<00:00, 48.52it/s]
100%|██████████| 835/835 [00:10<00:00, 83.31it/s]
100%|██████████| 836/836 [00:09<00:00, 83.82it/s]
```

(IMPLEMENTATION) Model Architecture

(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: My initial CNN architecture was to follow a similar style to Microsoft's ResNet and Oxford Univertities' VGG. That was to apply the same size filters in pairs repeatedly prior to reducing the parameters with pooling. Following this the plan was to add more than one dense layer at the output.

Initially and due to compute limitations I did this with two lots of 16 filters then a pool size of 2, followed by two lots of filters at 32 followed by a pool of 2. 16|16||2||32|32||2||

The dense layer I tried with a depth of one with the 133 outputs, this I found could be improved significantly by adding an additional dense layer of 500 features (with dropout), though due to its size the parameter size did increase significantly in size to over 100 million. halving and adding another dense layer would not be feasible.

I believe the volume of parameters at that point was too high. I then reverted back to something more simplistic. I removed the duplication of the convolution layers but maintained the dense layer of 500. This provided a parameter size of 50 million. By adding another convolutional layer with max pooling between this increases the filters used in detection and reduced total parameters to 25 Million. Though high, performance was reasonable for small to medium use. Additionally I did try reducing the 500 dense layer to a smaller value but this proved optimal and the additional compute delay was acceptable.

In [14]:

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

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
                         input_shape=(train_tensors[-1].shape)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu')
)
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu')
)
model.add(MaxPooling2D(pool_size=2))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(133, activation='softmax'))
### TODO: Define your architecture.

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 224, 224, 16)      208
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 112, 112, 32)      2080
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 64)        8256
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 28, 28, 64)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 50176)             0
_________________________________________________________________
dense_3 (Dense)              (None, 500)               25088500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               66633
=================================================================
Total params: 25,165,677
Trainable params: 25,165,677
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [15]:

```
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 [16]:

```
from keras.callbacks import ModelCheckpoint

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

epochs = 5

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

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

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.842
7 - acc: 0.0224Epoch 00001: val_loss improved from inf to 4.47213,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 33s 5ms/step - loss:
4.8416 - acc: 0.0225 - val_loss: 4.4721 - val_acc: 0.0623
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.247
1 - acc: 0.0727Epoch 00002: val_loss improved from 4.47213 to 4.18
731, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 32s 5ms/step - loss:
4.2461 - acc: 0.0728 - val_loss: 4.1873 - val_acc: 0.0814
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 3.521
8 - acc: 0.1898Epoch 00003: val_loss improved from 4.18731 to 4.16
156, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 32s 5ms/step - loss:
3.5209 - acc: 0.1898 - val_loss: 4.1616 - val_acc: 0.0898
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 2.365
4 - acc: 0.4198Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 32s 5ms/step - loss:
2.3668 - acc: 0.4193 - val_loss: 4.7915 - val_acc: 0.0814
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 1.257
1 - acc: 0.6809Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 32s 5ms/step - loss:
1.2581 - acc: 0.6808 - val_loss: 5.9884 - val_acc: 0.0898
```

Out[16]:

```
<keras.callbacks.History at 0x7f81b06677b8>
```

Load the Model with the Best Validation Loss

In [17]:

```
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 [18]:

```
# 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]
# display(dog_breed_predictions)

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

```
Test accuracy: 9.6890%
```

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 [19]:

```
bottleneck_features = np.load('/data/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 [20]:

```
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_5 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [21]:

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

Train the Model

In [22]:

```
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: 12.04
86 - acc: 0.1260Epoch 00001: val_loss improved from inf to 10.3632
0, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 3s 472us/step - loss:
12.0250 - acc: 0.1269 - val_loss: 10.3632 - val_acc: 0.2168
Epoch 2/20
6560/6680 [============================>.] - ETA: 0s - loss: 9.262
4 - acc: 0.3096Epoch 00002: val_loss improved from 10.36320 to 9.1
2951, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 3s 418us/step - loss:
9.2727 - acc: 0.3096 - val_loss: 9.1295 - val_acc: 0.3042
Epoch 3/20
```

```
6560/6680 [============================>.] - ETA: 0s - loss: 8.520
2 - acc: 0.3909Epoch 00003: val_loss improved from 9.12951 to 8.93
068, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 327us/step - loss:
8.5380 - acc: 0.3897 - val_loss: 8.9307 - val_acc: 0.3305
Epoch 4/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.313
3 - acc: 0.4287Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 2s 303us/step - loss:
8.3083 - acc: 0.4278 - val_loss: 8.9531 - val_acc: 0.3509
Epoch 5/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.178
2 - acc: 0.4515Epoch 00005: val_loss improved from 8.93068 to 8.80
174, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 312us/step - loss:
8.1807 - acc: 0.4513 - val_loss: 8.8017 - val_acc: 0.3713
Epoch 6/20
6600/6680 [============================>.] - ETA: 0s - loss: 8.111
7 - acc: 0.4652Epoch 00006: val_loss improved from 8.80174 to 8.76
932, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 319us/step - loss:
8.1042 - acc: 0.4656 - val_loss: 8.7693 - val_acc: 0.3808
Epoch 7/20
6640/6680 [============================>.] - ETA: 0s - loss: 8.047
1 - acc: 0.4777Epoch 00007: val_loss improved from 8.76932 to 8.74
559, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 311us/step - loss:
8.0465 - acc: 0.4775 - val_loss: 8.7456 - val_acc: 0.3737
Epoch 8/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.861
5 - acc: 0.4840Epoch 00008: val_loss improved from 8.74559 to 8.52
731, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 306us/step - loss:
7.8596 - acc: 0.4844 - val_loss: 8.5273 - val_acc: 0.3892
Epoch 9/20
6520/6680 [============================>.] - ETA: 0s - loss: 7.658
9 - acc: 0.4994Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 2s 317us/step - loss:
7.6540 - acc: 0.4994 - val_loss: 8.5717 - val_acc: 0.3892
Epoch 10/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.367
4 - acc: 0.5172Epoch 00010: val_loss improved from 8.52731 to 8.11
899, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 329us/step - loss:
7.3667 - acc: 0.5172 - val_loss: 8.1190 - val_acc: 0.4036
Epoch 11/20
6540/6680 [============================>.] - ETA: 0s - loss: 7.207
0 - acc: 0.5326Epoch 00011: val_loss improved from 8.11899 to 8.08
778, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 309us/step - loss:
7.2168 - acc: 0.5317 - val_loss: 8.0878 - val_acc: 0.4192
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 7.115
8 - acc: 0.5437Epoch 00012: val_loss improved from 8.08778 to 8.05
980, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 302us/step - loss:
7.1184 - acc: 0.5436 - val_loss: 8.0598 - val_acc: 0.4287
```

```
Epoch 13/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.092
8 - acc: 0.5489Epoch 00013: val_loss improved from 8.05980 to 7.92
837, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 306us/step - loss:
7.0918 - acc: 0.5490 - val_loss: 7.9284 - val_acc: 0.4383
Epoch 14/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.066
5 - acc: 0.5529Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 2s 303us/step - loss:
7.0682 - acc: 0.5528 - val_loss: 7.9663 - val_acc: 0.4395
Epoch 15/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.062
1 - acc: 0.5542Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 2s 303us/step - loss:
7.0582 - acc: 0.5543 - val_loss: 7.9305 - val_acc: 0.4323
Epoch 16/20
6660/6680 [============================>.] - ETA: 0s - loss: 6.972
2 - acc: 0.5554Epoch 00016: val_loss improved from 7.92837 to 7.81
765, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 310us/step - loss:
6.9827 - acc: 0.5548 - val_loss: 7.8177 - val_acc: 0.4419
Epoch 17/20
6620/6680 [============================>.] - ETA: 0s - loss: 6.780
1 - acc: 0.5672Epoch 00017: val_loss improved from 7.81765 to 7.73
671, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 313us/step - loss:
6.7903 - acc: 0.5665 - val_loss: 7.7367 - val_acc: 0.4503
Epoch 18/20
6660/6680 [============================>.] - ETA: 0s - loss: 6.627
2 - acc: 0.5769Epoch 00018: val_loss improved from 7.73671 to 7.52
073, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 313us/step - loss:
6.6315 - acc: 0.5766 - val_loss: 7.5207 - val_acc: 0.4719
Epoch 19/20
6600/6680 [============================>.] - ETA: 0s - loss: 6.516
8 - acc: 0.5867Epoch 00019: val_loss improved from 7.52073 to 7.51
527, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 322us/step - loss:
6.5237 - acc: 0.5862 - val_loss: 7.5153 - val_acc: 0.4635
Epoch 20/20
6640/6680 [============================>.] - ETA: 0s - loss: 6.506
7 - acc: 0.5914Epoch 00020: val_loss improved from 7.51527 to 7.46
868, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 2s 321us/step - loss:
6.4967 - acc: 0.5921 - val_loss: 7.4687 - val_acc: 0.4683
```

Out[22]:

```
<keras.callbacks.History at 0x7f81b01c2a58>
```

Load the Model with the Best Validation Loss

In [23]:

```
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 [24]:

```
# 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: 46.7703%
```

Predict Dog Breed with the Model

In [25]:

```
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. These are already in the workspace, at /data/bottleneck_features. If you wish to download them on a different machine, they can be found at:

- 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`.

The above architectures are downloaded and stored for you in the `/data/bottleneck_features/` folder.

This means the following will be in the `/data/bottleneck_features/` folder:

```
DogVGG19Data.npz  DogResnet50Data.npz  DogInceptionV3Data.npz
DogXceptionData.npz
```

(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('/data/bottleneck_features/Dog{network}Da
ta.npz')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

In [26]:

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features = np.load('/data/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: The model follows what I initially wanted to use in the manual CNN creation where I wanted to use a method similar to ResNet. By now using Transfer Learning this is now possible. With the output layer of ResNet removed we're using the GlobalAveragePooling to reduce the dimensionality. From here we're again having two dense layers with 500 features prior to the final output of 133 which classifies the dog. The number of parameters is significantly reduced to 1 million which is a great improvement over the manual CNN.

By having just one dense layer of 133 to classify the dog breed, this affected the accuracy. Adding multiple dense layers improved the accuracy. The final model choice with one additional dense layer of 500 features to the final 133 output layer I believe is an acceptable balance between compute time and accuracy.

In [27]:

```
### TODO: Define your architecture.
Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[1:]
))
Resnet50_model.add(Dense(500, activation='relu'))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_2 ( (None, 2048)              0
_________________________________________________________________
dense_6 (Dense)              (None, 500)               1024500
_________________________________________________________________
dense_7 (Dense)              (None, 133)               66633
=================================================================
Total params: 1,091,133
Trainable params: 1,091,133
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [28]:

```
### TODO: Compile the model.
Resnet50_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', m
etrics=['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 [29]:

```
### 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=5, batch_size=50, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6650/6680 [============================>.] - ETA: 0s - loss: 1.842
4 - acc: 0.5433Epoch 00001: val_loss improved from inf to 1.06003,
saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 2s 264us/step - loss:
1.8379 - acc: 0.5445 - val_loss: 1.0600 - val_acc: 0.6922
Epoch 2/5
6650/6680 [============================>.] - ETA: 0s - loss: 0.576
2 - acc: 0.8152Epoch 00002: val_loss improved from 1.06003 to 0.94
601, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s 175us/step - loss:
0.5754 - acc: 0.8153 - val_loss: 0.9460 - val_acc: 0.7257
Epoch 3/5
6600/6680 [============================>.] - ETA: 0s - loss: 0.322
8 - acc: 0.8967Epoch 00003: val_loss improved from 0.94601 to 0.92
305, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s 171us/step - loss:
0.3230 - acc: 0.8964 - val_loss: 0.9230 - val_acc: 0.7449
Epoch 4/5
6650/6680 [============================>.] - ETA: 0s - loss: 0.204
4 - acc: 0.9328Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 1s 166us/step - loss:
0.2046 - acc: 0.9328 - val_loss: 0.9436 - val_acc: 0.7569
Epoch 5/5
6500/6680 [============================>.] - ETA: 0s - loss: 0.152
7 - acc: 0.9515Epoch 00005: val_loss improved from 0.92305 to 0.85
661, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s 176us/step - loss:
0.1513 - acc: 0.9521 - val_loss: 0.8566 - val_acc: 0.8024
```

Out[29]:

```
<keras.callbacks.History at 0x7f81911259b0>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [30]:

```
### 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 [31]:

```
### 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: 79.1866%
```

(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 [32]:

```
### 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)])
```

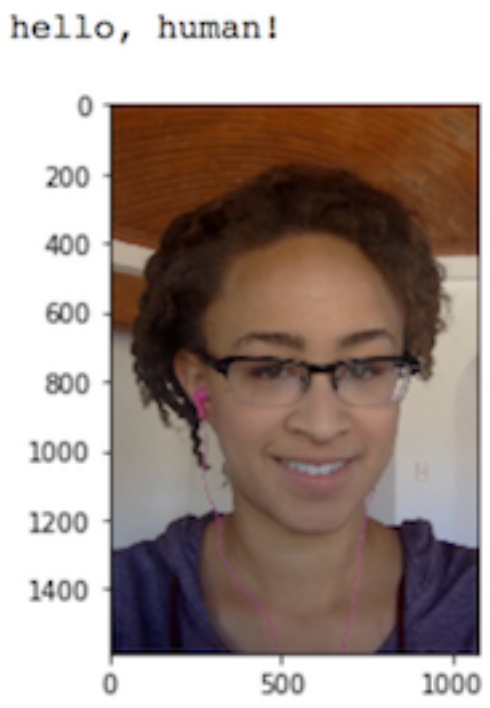
Step 6: Write your Algorithm

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 [33]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def Dog_breed_detector(img_path):
    #is the image of a human or a dog?
    if face_detector(img_path) == True and dog_detector(img_path) == False:
        image_type = 'Hello human!'
    elif face_detector(img_path) == False and dog_detector(img_path) == True:
        image_type = 'Hello dog!'
    else:
        #Either dogs and humans in there or none at all
        #both ways we'd need to throw an error
        image_type = None
    return image_type
```

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 results overall perform well. I placed a check that if a human or dog was not detected then it would provide a 'not sure' message. Similarly if an image was detected as having both a human and dog in there (such as we saw in the dog_files_sample) again it would display the same error. This could be an area for improvement where multiple images are seperated and each are uniquely identified. for example multiple humans, dogs or a mix are uniquely analysed.

The dog classifier only detects pedigree dogs i.e. those which are produced from a single breed type. If a dog is a cross-breed for example half English Cocker Spaniel and half Alsatian it may be possible to report on this. Since we're just reporting back the highest result from the 133 returns, if there is a close difference between the top 2 results that could help identify cross-breeds by ratio.

As we discussed previously, it could be possible to improve the human face detection by bearing in mind pictures where faces are not fully visible, or a picture of a profile face where half the features are missing.

I was curious to see if other images would be detected as humans or dogs by using images of neither. In these instances the detection looked to perform well being unable to identify a picture of a bee. I also tried to add ears and a dog nose superimposed on a human face which it was unable to identify. Similarly there were other features missing in the picture which would not identify the picture as a dog. I would suggest in this instance some features which identify a human (or dog) such as the nose are of significance. Removing the cartoon nose from the image then reported that the image was of a human with no other changes.

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.
import glob

def is_this_a_dog(imgs):
    i = 1
    fig = plt.figure(figsize=(50,50))
    for img in imgs:
        detector_type = Dog_breed_detector(img)
        if detector_type:
            xlabel_value = "You look like a..\n%s" %Resnet50_predict_breed(img
)[7:]
            display_image = cv2.imread(img)
            cv_rgb = cv2.cvtColor(display_image, cv2.COLOR_BGR2RGB)
            ax = fig.add_subplot(4, 3, i, xticks=[], yticks=[], title=detector
_type)
            ax.title.set_fontsize('32')
            ax.set_xlabel(xlabel_value, fontsize='32')
        else:
            #Didn't know if it was a Human or a Dog
            xlabel_value = "You look like a..\nsorry, I'm not sure!"
            display_image = cv2.imread(img)
            cv_rgb = cv2.cvtColor(display_image, cv2.COLOR_BGR2RGB)
            ax = fig.add_subplot(4, 3, i, xticks=[], yticks=[], title=detector
_type)
            ax.title.set_fontsize(32)
            ax.set_xlabel(xlabel_value, fontsize=32)
        ax.imshow(cv_rgb)
        i = i + 1

myfiles = glob.glob('images/samples/*.jpg')
is_this_a_dog(myfiles)
```

Hello human!

You look like a..
Basenji

Hello dog!

You look like a..
Poodle

Hello human!

You look like a..
Dachshund

None

You look like a..
sorry, I'm not sure!

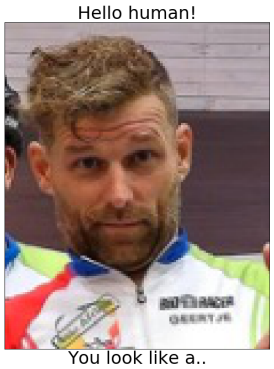
Hello dog!

You look like a..
Bearded_collie

Hello dog!

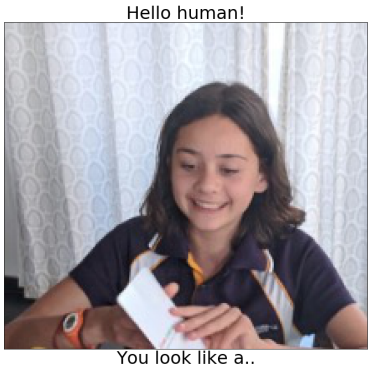
You look like a..
English_cocker_spaniel

Hello human!



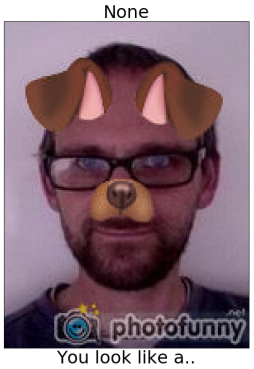
You look like a..
Australian_terrier

Hello human!



You look like a..
Italian_greyhound

None



You look like a..
sorry, I'm not sure!

Hello human!

You look like a..
Poodle

Hello dog!

You look like a..
Bulldog

Hello human!

You look like a..
Chinese_crested

Please download your notebook to submit

In order to submit, please do the following:

1. Download an HTML version of the notebook to your computer using 'File: Download as...'
2. Click on the orange Jupyter circle on the top left of the workspace.
3. Navigate into the dog-project folder to ensure that you are using the provided dog_images, lfw, and bottleneck_features folders; this means that those folders will *not* appear in the dog-project folder. If they do appear because you downloaded them, delete them.
4. While in the dog-project folder, upload the HTML version of this notebook you just downloaded. The upload button is on the top right.
5. Navigate back to the home folder by clicking on the two dots next to the folder icon, and then open up a terminal under the 'new' tab on the top right
6. Zip the dog-project folder with the following command in the terminal: `zip -r dog-project.zip dog-project`
7. Download the zip file by clicking on the square next to it and selecting 'download'. This will be the zip file you turn in on the next node after this workspace!