

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.

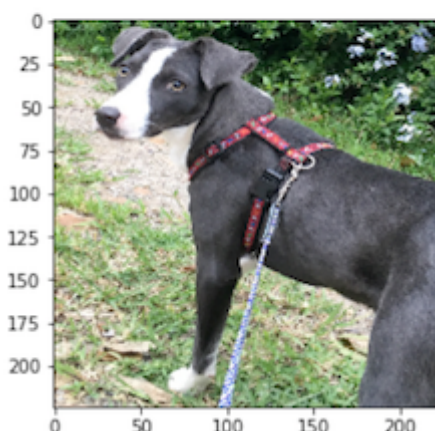
The rubric contains optional "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this IPython notebook.

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 [4]: 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 [5]: 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 [6]: import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

# load color (BGR) image
img = cv2.imread(human_files[21])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

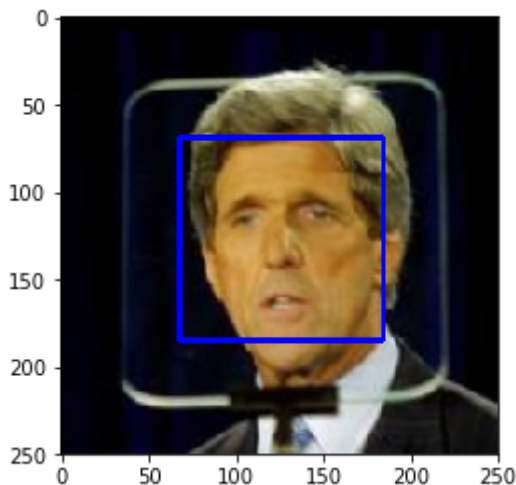
# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# 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 [7]: # 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: 98% of the first 100 images in `human_files` have a detected human face. While 11% of the first 100 dog images from `dog_files` have a detected human face.

```
In [8]: 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
total_human_accuracy = 0
total_dog_accuracy = 0
for x in range(len(human_files_short)):
    if(face_detector(human_files_short[x])):
        total_human_accuracy+=1
for x in range(len(dog_files_short)):
    if(face_detector(dog_files_short[x])):
        total_dog_accuracy +=1

print("Total human accuracy = {}/{} which is {:.2f}%".format(total_human_acc
print("Total dog accuracy = {}/{} which is {:.2f}%".format(total_dog_accurac

## on the images in human_files_short and dog_files_short.
```

```
Total human accuracy = 98/100 which is 98.00%
Total dog accuracy = 11/100 which is 11.00%
```

Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having 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?

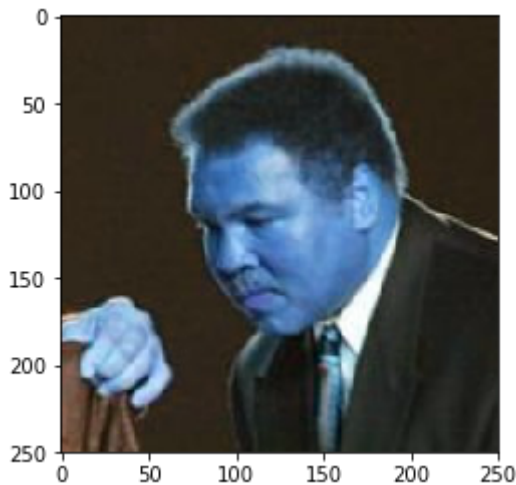
Answer: I think it turns user off if you restrict image upload to be human face only. A better approach would be providing a generic error msg when no human face's detected. As an example: "No human face detected. Possible reason might includes (not limited to) blurry picture or picture does not include human face."

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 [12]: ## (Optional) TODO: Report the performance of another
#https://github.com/ageitgey/face_recognition#face-recognition

import face_recognition
print(human_files_short[0])
image = face_recognition.load_image_file(human_files_short[0])
img = cv2.imread(human_files_short[0])
plt.imshow(img)
plt.show()
face_landmarks_list = face_recognition.face_landmarks(image)
print(face_landmarks_list)
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.
```

lfw/Muhammad_Ali/Muhammad_Ali_0001.jpg



```
[{'left_eyebrow': [(80, 117), (80, 114), (82, 113), (84, 114), (87, 114), (87, 111), (84, 110), (82, 109), (80, 108), (78, 107), (76, 106), (74, 105), (72, 104), (70, 103), (68, 102), (66, 101), (64, 100), (62, 99), (60, 98), (58, 97), (56, 96), (54, 95), (52, 94), (50, 93), (48, 92), (46, 91), (44, 90), (42, 89), (40, 88), (38, 87), (36, 86), (34, 85), (32, 84), (30, 83), (28, 82), (26, 81), (24, 80), (22, 79), (20, 78), (18, 77), (16, 76), (14, 75), (12, 74), (10, 73), (8, 72), (6, 71), (4, 70), (2, 69), (0, 68), (0, 67), (0, 66), (0, 65), (0, 64), (0, 63), (0, 62), (0, 61), (0, 60), (0, 59), (0, 58), (0, 57), (0, 56), (0, 55), (0, 54), (0, 53), (0, 52), (0, 51), (0, 50), (0, 49), (0, 48), (0, 47), (0, 46), (0, 45), (0, 44), (0, 43), (0, 42), (0, 41), (0, 40), (0, 39), (0, 38), (0, 37), (0, 36), (0, 35), (0, 34), (0, 33), (0, 32), (0, 31), (0, 30), (0, 29), (0, 28), (0, 27), (0, 26), (0, 25), (0, 24), (0, 23), (0, 22), (0, 21), (0, 20), (0, 19), (0, 18), (0, 17), (0, 16), (0, 15), (0, 14), (0, 13), (0, 12), (0, 11), (0, 10), (0, 9), (0, 8), (0, 7), (0, 6), (0, 5), (0, 4), (0, 3), (0, 2), (0, 1), (0, 0)], 'top_lip': [(101, 161), (99, 158), (100, 158), (102, 158), (106, 156), (114, 156), (123, 156), (121, 157), (107, 159), (104, 160), (101, 160), (102, 161)], 'chin': [(89, 129), (89, 136), (92, 144), (95, 152), (99, 161), (102, 169), (106, 177), (111, 185), (119, 187), (131, 184), (143, 178), (154, 171), (164, 161), (170, 149), (171, 134), (169, 119), (166, 105)], 'right_eye': [(110, 119), (112, 117), (117, 116), (123, 116), (118, 118), (113, 119)], 'bottom_lip': [(123, 156), (117, 162), (111, 165), (107, 166), (104, 166), (102, 165), (101, 161), (102, 161), (102, 161), (105, 161), (108, 160), (121, 157)], 'nose_bridge': [(94, 121), (93, 128), (92, 135), (91, 142)], 'right_eyebrow': [(96, 112), (103, 108), (112, 106), (121, 106), (131, 108)], 'left_eye': [(87, 124), (87, 122), (90, 121), (94, 122), (91, 123), (88, 124)], 'nose_tip': [(91, 147), (94, 149), (98, 149), (103, 147), (108, 145)]}]
```

```
In [15]: def face_detector2(img_path):
image = face_recognition.load_image_file(img_path)
face_landmarks_list = face_recognition.face_landmarks(image)
return len(face_landmarks_list) > 0
```



```
In [16]: total_human_accuracy = 0
total_dog_accuracy = 0
for x in range(len(human_files_short)):
    if(face_detector2(human_files_short[x])):
        total_human_accuracy+=1
for x in range(len(dog_files_short)):
    if(face_detector2(dog_files_short[x])):
        total_dog_accuracy +=1

print("Total human accuracy = {}/{} which is {:.2f}%".format(total_human_acc
print("Total dog accuracy = {}/{} which is {:.2f}%".format(total_dog_accura
```

```
Total human accuracy = 100/100 which is 100.00%
Total dog accuracy = 10/100 which is 10.00%
```

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 [17]: from keras.applications.resnet50 import ResNet50
```

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

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels.h5 (https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels.h5)
```

Pre-process the Data

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

$$[nb_samples, nb_rows, nb_columns, nb_channels]$$

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 [18]: 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
            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 $[127.5, 127.5, 127.5]$ 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) (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 `g`th entry is the model's predicted probability that the image belongs to the `g`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) (<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>).

```
In [19]: from keras.applications.resnet50 import preprocess_input, decode_predictions

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

Write a Dog Detector

While looking at the [dictionary \(https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a\)](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 [20]: ### 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

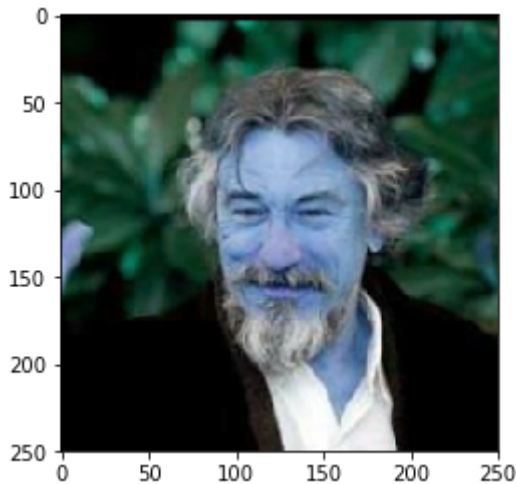
Question 3: Use the code cell below to test the performance of your dog_detector function.

- 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: Total human accuracy = 1/100 which is 1.00%. Total dog accuracy = 100/100 which is 100.00%.

```
In [22]: ### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
total_human_accuracy = 0
total_dog_accuracy = 0
for x in range(len(human_files_short)):
    if(dog_detector(human_files_short[x])):
        total_human_accuracy+=1
        #curious on which human got identified as dog...
        img = cv2.imread(human_files_short[x])
        plt.imshow(img)
        plt.show()
for x in range(len(dog_files_short)):
    if(dog_detector(dog_files_short[x])):
        total_dog_accuracy +=1

print("Total human accuracy = {}/{} which is {:.2f}%".format(total_human_acc
print("Total dog accuracy = {}/{} which is {:.2f}%".format(total_dog_accurac
```



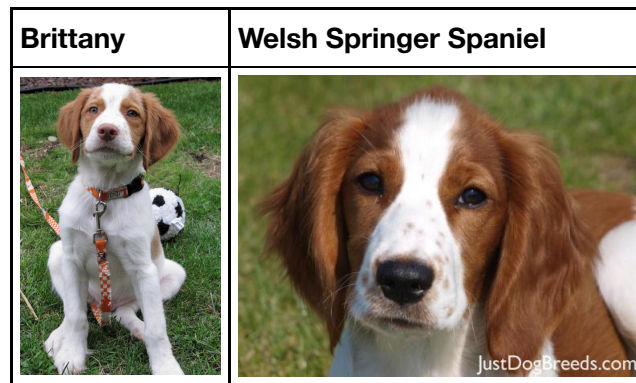
```
Total human accuracy = 1/100 which is 1.00%
Total dog accuracy = 100/100 which is 100.00%
```

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

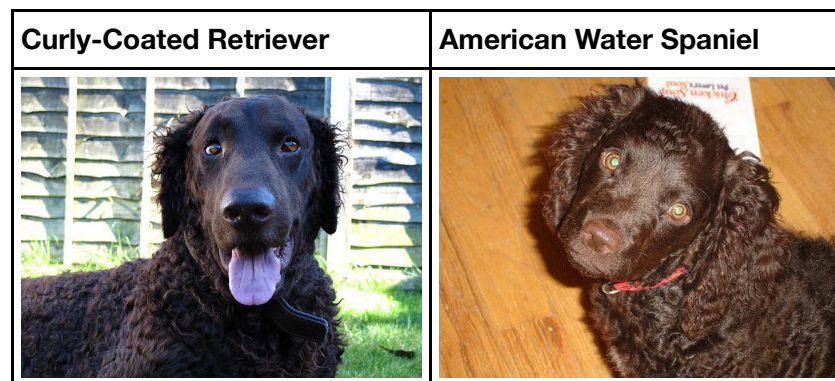
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN from scratch (so, you can't use transfer learning yet!), and you must attain a test accuracy of at least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

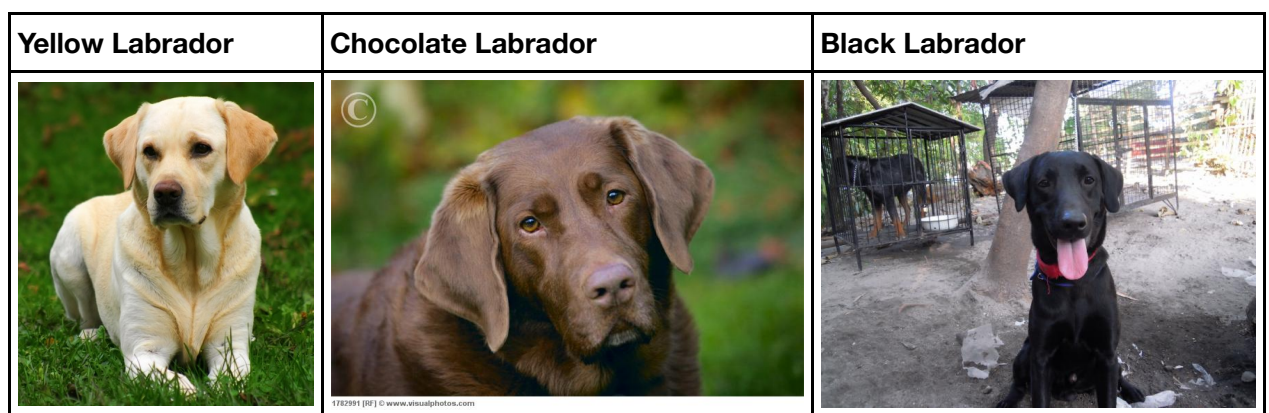
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that even a human would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.



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.



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 [23]: 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 [01:58<00:00, 56.57it/s]
100%|██████████| 835/835 [00:13<00:00, 71.69it/s]
100%|██████████| 836/836 [00:13<00:00, 62.35it/s]
```

(IMPLEMENTATION) Model Architecture

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

```
model.summary()
```

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

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 223, 223, 16)	208
max_pooling2d_1 (MaxPooling2D)	(None, 111, 111, 16)	0
conv2d_2 (Conv2D)	(None, 110, 110, 32)	2080
max_pooling2d_2 (MaxPooling2D)	(None, 55, 55, 32)	0
conv2d_3 (Conv2D)	(None, 54, 54, 64)	8256
max_pooling2d_3 (MaxPooling2D)	(None, 27, 27, 64)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 64)	0
dense_1 (Dense)	(None, 133)	8645
Total params: 19,189.0		
Trainable params: 19,189.0		
Non-trainable params: 0.0		

INPUT

CONV

POOL

CONV

POOL

CONV


POOL

GAP

DENSE

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:

Type Markdown and LaTeX: 

```
In [57]: ##
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(16, kernel_size=(1, 1), strides=(1, 1),
                 activation='relu', padding='same',
                 input_shape=(224, 224, 3)))

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(GlobalAveragePooling2D(data_format='None'))
model.add(Dense(133, activation='softmax'))
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_45 (Conv2D)	(None, 224, 224, 16)	64

max_pooling2d_29 (MaxPooling)	(None, 112, 112, 16)	0

conv2d_46 (Conv2D)	(None, 112, 112, 32)	2080

max_pooling2d_30 (MaxPooling)	(None, 56, 56, 32)	0

conv2d_47 (Conv2D)	(None, 56, 56, 64)	8256

max_pooling2d_31 (MaxPooling)	(None, 28, 28, 64)	0

global_average_pooling2d_6 (GlobalAveragePooling2D)	(None, 64)	0

dense_5 (Dense)	(None, 133)	8645
=====		
Total params: 19,045.0		
Trainable params: 19,045.0		
Non-trainable params: 0.0		

Compile the Model

```
In [58]: 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 [59]: from keras.callbacks import ModelCheckpoint

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

epochs = 10

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

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

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

Train on 6680 samples, validate on 835 samples

Epoch 1/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.8826 - acc: 0.0107

model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 163s - loss: 4.8824 - acc: 0.0106 - val_loss: 4.8663 - val_acc: 0.0108

Epoch 2/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.8559 - acc: 0.0104

model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 146s - loss: 4.8559 - acc: 0.0103 - val_loss: 4.8379 - val_acc: 0.0192

Epoch 3/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.8152 - acc: 0.0179

model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 147s - loss: 4.8153 - acc: 0.0178 - val_loss: 4.8082 - val_acc: 0.0192

Epoch 4/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.7851 - acc: 0.0188

saving model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 149s - loss: 4.7852 - acc: 0.0189 - val_loss: 4.7990 - val_acc: 0.0228

Epoch 5/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.7645 - acc: 0.0212

Epoch 00004: val_loss improved from 4.79904 to 4.78553, saving model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 148s - loss: 4.7645 - acc: 0.0213 - val_loss: 4.7855 - val_acc: 0.0204

Epoch 6/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.7454 - acc: 0.0233

saving model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 142s - loss: 4.7452 - acc: 0.0232 - val_loss: 4.7805 - val_acc: 0.0216

Epoch 7/10

6660/6680 [=====>.] - ETA: 0s - loss: 4.7283 - acc: 0.0245

model to saved_models/weights.best.from_scratch.hdf5

6680/6680 [=====] - 145s - loss: 4.7281 - acc:

```

0.0244 - val_loss: 4.7534 - val_acc: 0.0263
Epoch 8/10
6660/6680 [=====>.] - ETA: 0s - loss: 4.7085 - acc: 0.0272
model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 146s - loss: 4.7083 - acc: 0.0271 - val_loss: 4.7484 - val_acc: 0.0263
Epoch 9/10
6660/6680 [=====>.] - ETA: 0s - loss: 4.6890 - acc: 0.0294
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 150s - loss: 4.6892 - acc: 0.0293 - val_loss: 4.7253 - val_acc: 0.0299
Epoch 10/10
6660/6680 [=====>.] - ETA: 0s - loss: 4.6704 - acc: 0.0315 Epoch 00009: val_loss improved from 4.72530 to 4.71635, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [=====] - 148s - loss: 4.6706 - acc: 0.0314 - val_loss: 4.7164 - val_acc: 0.0347

```

```
Out[59]: <keras.callbacks.History at 0x125d14860>
```

Load the Model with the Best Validation Loss

```
In [60]: 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 [61]: # 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)), axis=-1) for tensor in test_images]

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

```
Test accuracy: 3.3493%
```

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

Model Architecture

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

```
In [63]: 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_7 ((None, 512)		0
dense_6 (Dense)	(None, 133)	68229
Total params: 68,229.0		
Trainable params: 68,229.0		
Non-trainable params: 0.0		

Compile the Model

```
In [64]: VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', me
```

Train the Model

```
In [65]: 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=[checker], verbose=1)
```

Train on 6680 samples, validate on 835 samples

Epoch 1/20

6600/6680 [=====>.] - ETA: 0s - loss: 12.6221 - acc: 0.1009 Epoch 00000: val_loss improved from inf to 11.01860, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 1s - loss: 12.5986 - acc: 0.1021 - val_loss: 11.0186 - val_acc: 0.2072

Epoch 2/20

6300/6680 [=====>..] - ETA: 0s - loss: 10.4568 - acc: 0.2589 Epoch 00001: val_loss improved from 11.01860 to 10.54362, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 10.4716 - acc: 0.2588 - val_loss: 10.5436 - val_acc: 0.2551

Epoch 3/20

6320/6680 [=====>..] - ETA: 0s - loss: 10.0690 - acc: 0.3127 Epoch 00002: val_loss improved from 10.54362 to 10.42127, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 10.0542 - acc: 0.3139 - val_loss: 10.4213 - val_acc: 0.2826

Epoch 4/20

6520/6680 [=====>.] - ETA: 0s - loss: 9.8327 - acc: 0.3443 Epoch 00003: val_loss improved from 10.42127 to 10.25515, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 9.8230 - acc: 0.3446 - val_loss: 10.2552 - val_acc: 0.2970

Epoch 5/20

6500/6680 [=====>.] - ETA: 0s - loss: 9.7310 - acc: 0.3637 Epoch 00004: val_loss improved from 10.25515 to 10.16613, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 9.7086 - acc: 0.3645 - val_loss: 10.1661 - val_acc: 0.2958

Epoch 6/20

6560/6680 [=====>.] - ETA: 0s - loss: 9.5958 - acc: 0.3730 Epoch 00005: val_loss improved from 10.16613 to 10.00418, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 1s - loss: 9.5870 - acc: 0.3731 - val_loss: 10.0042 - val_acc: 0.3186

Epoch 7/20

6360/6680 [=====>..] - ETA: 0s - loss: 9.3502 - acc: 0.3888 Epoch 00006: val_loss improved from 10.00418 to 9.86203, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 9.3230 - acc: 0.3891 - val_loss: 9.8620 - val_acc: 0.3174

Epoch 8/20

6320/6680 [=====>..] - ETA: 0s - loss: 9.0265 - acc: 0.4073 Epoch 00007: val_loss improved from 9.86203 to 9.64901, saving model to saved_models/weights.best.VGG16.hdf5

6680/6680 [=====] - 0s - loss: 9.0336 - acc: 0.4076 - val_loss: 9.6490 - val_acc: 0.3305

Epoch 9/20

```
6580/6680 [=====>.] - ETA: 0s - loss: 8.9040 - acc: 0.4217 Epoch 00008: val_loss improved from 9.64901 to 9.51155, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.8810 - acc: 0.4231 - val_loss: 9.5116 - val_acc: 0.3485
Epoch 10/20
6280/6680 [=====>..] - ETA: 0s - loss: 8.7536 - acc: 0.4358 Epoch 00009: val_loss improved from 9.51155 to 9.49583, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.7407 - acc: 0.4368 - val_loss: 9.4958 - val_acc: 0.3461
Epoch 11/20
6440/6680 [=====>..] - ETA: 0s - loss: 8.6804 - acc: 0.4452 Epoch 00010: val_loss improved from 9.49583 to 9.41771, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 0s - loss: 8.6692 - acc: 0.4461 - val_loss: 9.4177 - val_acc: 0.3413
Epoch 12/20
6580/6680 [=====>.] - ETA: 0s - loss: 8.5834 - acc: 0.4517 Epoch 00011: val_loss improved from 9.41771 to 9.35112, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.5733 - acc: 0.4525 - val_loss: 9.3511 - val_acc: 0.3629
Epoch 13/20
6540/6680 [=====>.] - ETA: 0s - loss: 8.4397 - acc: 0.4619 Epoch 00012: val_loss improved from 9.35112 to 9.22667, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.4406 - acc: 0.4621 - val_loss: 9.2267 - val_acc: 0.3605
Epoch 14/20
6580/6680 [=====>.] - ETA: 0s - loss: 8.2472 - acc: 0.4734 Epoch 00013: val_loss improved from 9.22667 to 9.10800, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 8.2565 - acc: 0.4729 - val_loss: 9.1080 - val_acc: 0.3629
Epoch 15/20
6340/6680 [=====>..] - ETA: 0s - loss: 7.8712 - acc: 0.4877 Epoch 00014: val_loss improved from 9.10800 to 8.58003, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.8844 - acc: 0.4876 - val_loss: 8.5800 - val_acc: 0.3892
Epoch 16/20
6580/6680 [=====>.] - ETA: 0s - loss: 7.6203 - acc: 0.5064 Epoch 00015: val_loss improved from 8.58003 to 8.43753, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.6410 - acc: 0.5049 - val_loss: 8.4375 - val_acc: 0.4096
Epoch 17/20
6480/6680 [=====>.] - ETA: 0s - loss: 7.6149 - acc: 0.5148 Epoch 00016: val_loss did not improve
6680/6680 [=====] - 1s - loss: 7.5831 - acc: 0.5169 - val_loss: 8.4778 - val_acc: 0.4060
Epoch 18/20
6500/6680 [=====>.] - ETA: 0s - loss: 7.5624 - acc: 0.5203 Epoch 00017: val_loss improved from 8.43753 to 8.39937, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.5480 - acc: 0.5203
```

```

211 - val_loss: 8.3994 - val_acc: 0.4084
Epoch 19/20
6500/6680 [=====>.] - ETA: 0s - loss: 7.4662 - ac
c: 0.5265Epoch 00018: val_loss improved from 8.39937 to 8.29639, saving m
odel to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.4742 - acc: 0.5
259 - val_loss: 8.2964 - val_acc: 0.4144
Epoch 20/20
6500/6680 [=====>.] - ETA: 0s - loss: 7.3339 - ac
c: 0.5345Epoch 00019: val_loss improved from 8.29639 to 8.14860, saving m
odel to saved_models/weights.best.VGG16.hdf5
6680/6680 [=====] - 1s - loss: 7.3037 - acc: 0.5
361 - val_loss: 8.1486 - val_acc: 0.4287

```

```
Out[65]: <keras.callbacks.History at 0x124f28e10>
```

Load the Model with the Best Validation Loss

```
In [66]: 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 [67]: # get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, a

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

Test accuracy: 43.6603%
```

Predict Dog Breed with the Model

```
In [68]: 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 [74]: ### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features19 = np.load('bottleneck_features/DogVGG19Data.npz')
train_vgg19 = bottleneck_features19['train']
valid_vgg19 = bottleneck_features19['valid']
test_vgg19 = bottleneck_features19['test']
```

(IMPLEMENTATION) Model Architecture

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

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

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

Answer: I tried to mimic the example on VGG16 but increase the epochs and defined a learning rate. With some trial and error, I was able to achieve above 60% of accuracy against the test data set. This architecture is suitable for the current problem because it's efficient enough as it doesn't take whole lot of time to train.

```
In [76]: ### TODO: Define your architecture.
VGG19_model = Sequential()
VGG19_model.add(GlobalAveragePooling2D(input_shape=train_vgg19.shape[1:]))

VGG19_model.add(Dense(133, activation='softmax'))

VGG19_model.summary()
```

Layer (type)	Output Shape	Param #
global_average_pooling2d_8 ((None, 512)		0
dense_7 (Dense)	(None, 133)	68229
Total params: 68,229.0		
Trainable params: 68,229.0		
Non-trainable params: 0.0		

(IMPLEMENTATION) Compile the Model

```
In [97]: ### TODO: Compile the model.
#VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', r
from keras import optimizers
adam = optimizers.Adam(lr=0.001)
VGG19_model.compile(loss='categorical_crossentropy', optimizer=adam, metrics
```

(IMPLEMENTATION) Train the Model

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

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


```
In [104]: ### TODO: Train the model.
#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)

checkpointers = ModelCheckpoint(filepath='saved_models/weights.best.VGG19.hdf5',
                                verbose=1, save_best_only=True)
VGG19_model.fit(train_vgg19, train_targets, validation_data=(valid_vgg19, valid_targets),
                epochs=60, batch_size=20, callbacks=[checkpointers], verbose=1)
```

Train on 6680 samples, validate on 835 samples

Epoch 1/60

6560/6680 [=====>.] - ETA: 0s - loss: 3.9042 - acc: 0.7578Epoch 00000: val_loss improved from inf to 5.61409, saving model to saved_models/weights.best.VGG19.hdf5

6680/6680 [=====] - 1s - loss: 3.9041 - acc: 0.7578 - val_loss: 5.6141 - val_acc: 0.5844

Epoch 2/60

6580/6680 [=====>.] - ETA: 0s - loss: 3.8948 - acc: 0.7584Epoch 00001: val_loss improved from 5.61409 to 5.61269, saving model to saved_models/weights.best.VGG19.hdf5

6680/6680 [=====] - 1s - loss: 3.9041 - acc: 0.7578 - val_loss: 5.6127 - val_acc: 0.5868

Epoch 3/60

6600/6680 [=====>.] - ETA: 0s - loss: 3.8903 - acc: 0.7586Epoch 00002: val_loss improved from 5.61269 to 5.61259, saving model to saved_models/weights.best.VGG19.hdf5

6680/6680 [=====] - 1s - loss: 3.9041 - acc: 0.7578 - val_loss: 5.6126 - val_acc: 0.5856

Epoch 4/60

(IMPLEMENTATION) Load the Model with the Best Validation Loss

```
In [105]: ### TODO: Load the model weights with the best validation loss.
#VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
VGG19_model.load_weights('saved_models/weights.best.VGG19.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 [106]: ### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
#VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, a

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

VGG19_predictions = [np.argmax(VGG19_model.predict(np.expand_dims(feature, a

test_accuracy = 100*np.sum(np.array(VGG19_predictions) == np.argmax(test_tar
print('Test accuracy: %.4f%%' % test_accuracy)

Test accuracy: 60.2871%
```

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

def VGG19_predict_breed(img_path):
    bottleneck_feature = extract_VGG19(path_to_tensor(img_path))
    predicted_vector = VGG19_model.predict(bottleneck_feature)
    return dog_names[np.argmax(predicted_vector)]
```

```
---
<a id='step6'></a>
## 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!

![[Sample Human Output](images/sample_human_output.png)]

(IMPLEMENTATION) Write your Algorithm

```
In [115]: ### TODO: Write your algorithm.
def which_dog_is_this(image_path):
    is_dog = dog_detector(image_path)
    is_human = face_detector2(image_path)
    if(is_human and is_dog):
        top_phrase = "hello!"
    elif(is_human): #if human is on, then human
        top_phrase = "hello, human!"
    else:
        top_phrase = "hello, dog!"
    breed_prediction = VGG19_predict_breed(image_path)
    print(top_phrase)
    img = cv2.imread(image_path)
    plt.imshow(img)
    plt.show()
    print("You look like a...")
    print(breed_prediction)
```

Feel free to use as many code cells as needed.

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

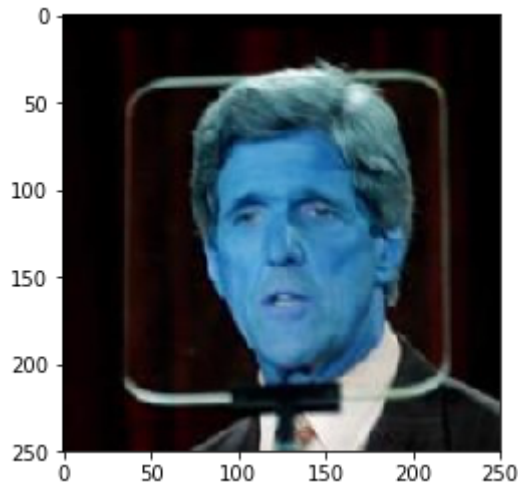
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: The output for predicting dog is better than what I expected. However, the prediction on what type of dog breed does a human look like did not work as well. I have listed three point of improvement below:

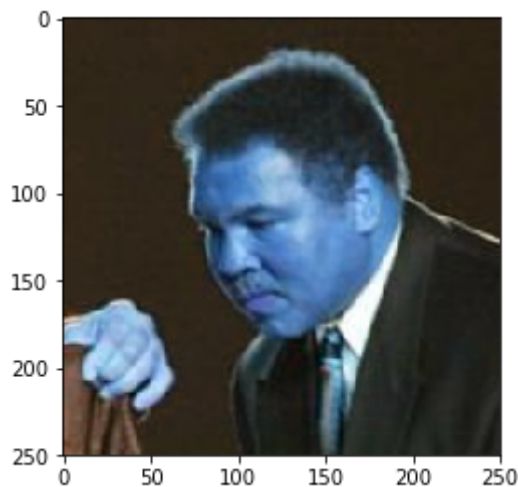
1. Altered the CNN network architect to achieve higher % of accuracy (80%)?
2. Change the test input image for human to at least cover the upper body (consistent upper body and face ratio).
3. Convert all images to gray scale.

```
In [116]: ## 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.
my_image = human_files[21]
which_dog_is_this(my_image)
my_image = human_files_short[0]
which_dog_is_this(my_image)
```

hello, human!



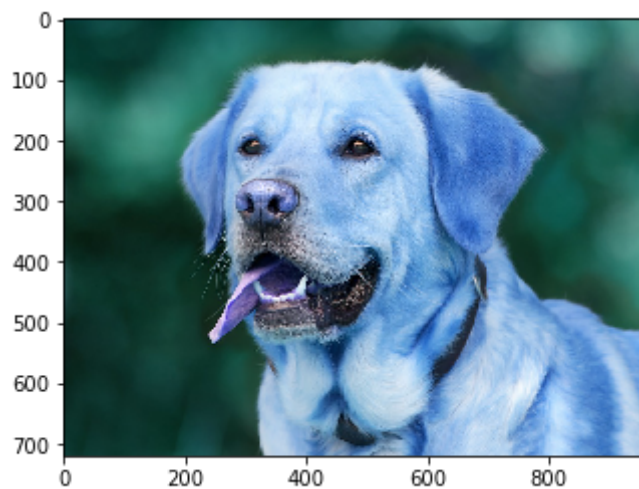
You look like a...
Italian_greyhound
hello, human!



You look like a...
French_bulldog

```
In [117]: step_7_files = np.array(glob("step7_images/*"))  
          for x in range(len(step_7_files)):  
              my_image = step_7_files[x]  
              which_dog_is_this(my_image)
```

hello!



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

In []: