

Traffic_Sign_Classifier_Project2

July 15, 2017

1 Self-Driving Car Engineer Nanodegree

1.1 Deep Learning

1.2 Project: Build a Traffic Sign Recognition Classifier

In this notebook, a template is provided for you to implement your functionality in stages, which is required to successfully complete this project. If additional code is required that cannot be included in the notebook, be sure that the Python code is successfully imported and included in your submission if necessary.

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

In addition to implementing code, there is a writeup to complete. The writeup should be completed in a separate file, which can be either a markdown file or a pdf document. There is a [write up template](#) that can be used to guide the writing process. Completing the code template and writeup template will cover all of the [rubric points](#) for this project.

The [rubric](#) contains "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. The stand out suggestions are optional. If you decide to pursue the "stand out suggestions", you can include the code in this Ipython notebook and also discuss the results in the writeup file.

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

```
In [1]: import os
        os.getcwd()
```

```
Out[1]: '/Users/mac/CarND-Traffic-Sign-Classifer-Project/Traffic-Sign-Classifer-Project'
```

```
In [2]: import csv

        with open('signnames.csv', 'r') as csvfile:
            reader = csv.reader(csvfile)
            labels_dict= {x: y for x,y in reader}

        labels_dict.pop('ClassId')
        print('Labels Dictionary Created')

        Labels Dictionary Created
```

1.3 Step 0: Load The Data

```
In [3]: # Load pickled data
import pickle

# TODO: Fill this in based on where you saved the training and testing data

training_file = './traffic-signs-data/train.p'
validation_file='./traffic-signs-data/valid.p'
testing_file = './traffic-signs-data/test.p'

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(validation_file, mode='rb') as f:
    valid = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

1.4 Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- 'features' is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- 'labels' is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- 'sizes' is a list containing tuples, (width, height) representing the original width and height the image.
- 'coords' is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the results. For example, the [pandas shape method](#) might be useful for calculating some of the summary results.

1.4.1 Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

```
In [4]: ### Replace each question mark with the appropriate value.
        ### Use python, pandas or numpy methods rather than hard coding the results

# TODO: Number of training examples
n_train = X_train.shape[0]

# TODO: Number of validation examples
n_validation = X_valid.shape[0]

# TODO: Number of testing examples.
n_test = X_test.shape[0]
```

```

# TODO: What's the shape of an traffic sign image?
image_shape = X_train[0].shape

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(set(y_train))

print("Number of training examples =", n_train)
print("Number of validation examples =", n_validation)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)

Number of training examples = 34799
Number of validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43

```

1.4.2 Include an exploratory visualization of the dataset

Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

The [Matplotlib examples](#) and [gallery](#) pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. It can be interesting to look at the distribution of classes in the training, validation and test set. Is the distribution the same? Are there more examples of some classes than others?

```

In [5]: ### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random
# Visualizations will be shown in the notebook.
%matplotlib inline
index = random.randint(0,n_train)
sample = X_train[index]
sample_label = y_train[index]
print('Sample Image\nLabel # {}'.format(sample_label,labels_dict[str(sample_label)]))
plt.imshow(sample)

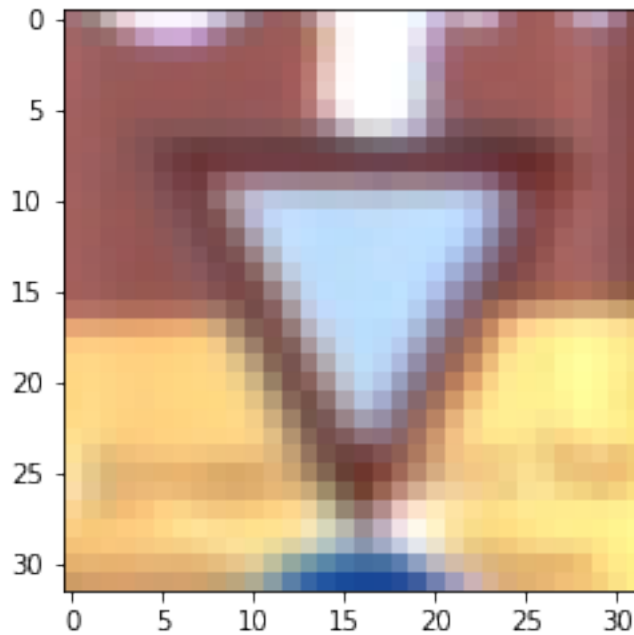
Sample Image
Label # 13      Yield

```

```

Out[5]: <matplotlib.image.AxesImage at 0x120095780>

```



1.5 Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the [German Traffic Sign Dataset](#).

The LeNet-5 implementation shown in the [classroom](#) at the end of the CNN lesson is a solid starting point. You'll have to change the number of classes and possibly the preprocessing, but aside from that it's plug and play!

With the LeNet-5 solution from the lecture, you should expect a validation set accuracy of about 0.89. To meet specifications, the validation set accuracy will need to be at least 0.93. It is possible to get an even higher accuracy, but 0.93 is the minimum for a successful project submission.

There are various aspects to consider when thinking about this problem:

- Neural network architecture (is the network over or underfitting?)
- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

Here is an example of a [published baseline model on this problem](#). It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

1.5.1 Pre-process the Data Set (normalization, grayscale, etc.)

Minimally, the image data should be normalized so that the data has mean zero and equal variance. For image data, $(\text{pixel} - 128) / 128$ is a quick way to approximately normalize the data and can be used in this project.

Other pre-processing steps are optional. You can try different techniques to see if it improves performance.

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project.

```
In [6]: ### Preprocess the data here. It is required to normalize the data.  
       ### Other preprocessing steps could include:  
       ### converting to grayscale, etc.  
       ### Feel free to use as many code cells as needed.  
import numpy as np  
import cv2
```

```

##GrayScale
def grayscale(X):
    X_gray = np.zeros([X.shape[0],32,32,1])
    for i in range(X.shape[0]):
        img = cv2.cvtColor(X[i], cv2.COLOR_RGB2GRAY)
        img = cv2.equalizeHist(img)
        X_gray[i] = np.expand_dims(img, axis=2)
    return X_gray

##Normalize with Min-Max
def normalize(X):
    X_norm = np.zeros_like(X, dtype=np.float32)
    for i in range(X.shape[0]):
        X_norm[i] = cv2.normalize(X[i], X_norm[i],
                                alpha=-1., beta=1.,
                                norm_type=cv2.NORM_MINMAX,
                                dtype=cv2.CV_32F)

    return X_norm

```

In [7]: *#GrayScale and Normalize Training Set*

```

X_gray = grayscale(X_train)
X_norm = normalize(X_gray)

```

#GrayScale and Normalize Validation Set

```

X_valid_gray = grayscale(X_valid)
X_valid_gray = normalize(X_valid_gray)

```

#GrayScale and Normalize Test Set

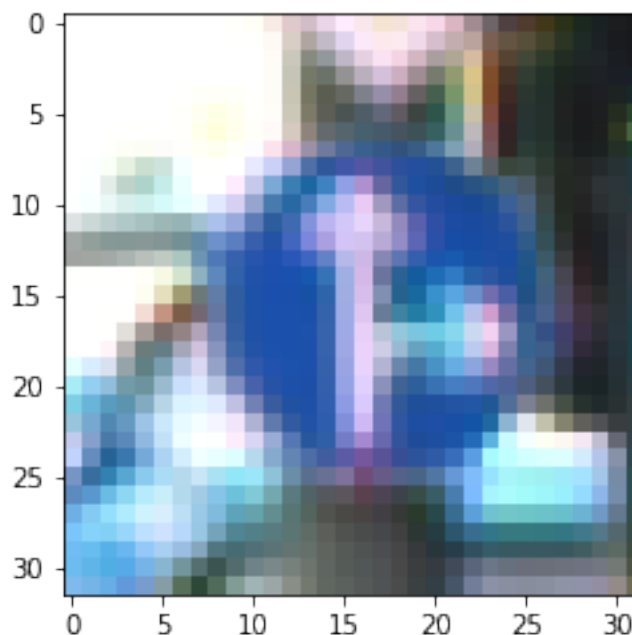
```

X_test_gray = grayscale(X_test)
X_test_gray = normalize(X_test_gray)

```

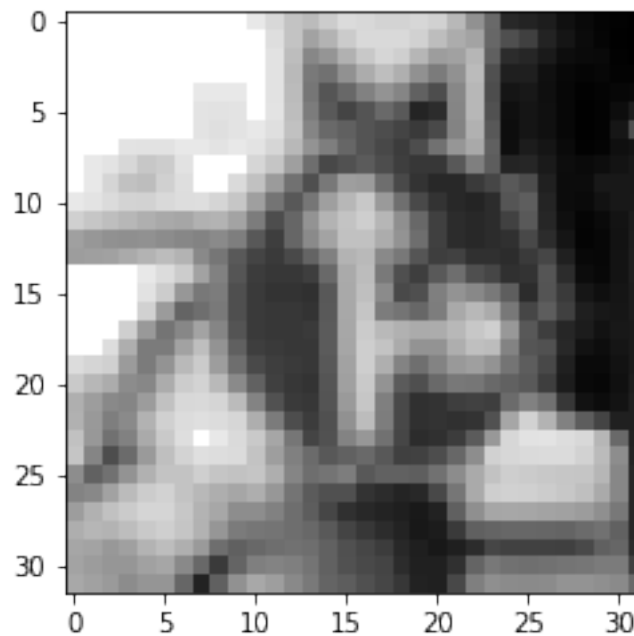
In [8]: plt.imshow(X_train[1020])

Out[8]: <matplotlib.image.AxesImage at 0x141830a58>



```
In [9]: plt.imshow(X_norm[1020].reshape(32,32), cmap='gray')
```

```
Out[9]: <matplotlib.image.AxesImage at 0x141865ac8>
```



```
In [10]: #Shuffle
         from sklearn.utils import shuffle

         X_train_S, y_train_S = shuffle(X_norm, y_train)
```

1.5.2 Model Architecture

```
In [11]: ### Define your architecture here.
         ### Feel free to use as many code cells as needed.
         import tensorflow as tf

         tf.reset_default_graph()

         #Model Parameters Set
         EPOCHS = 15
         BATCH_SIZE = 128
         learn_rate = 0.001
         mean = 0.
         standard_dev = 0.1
         dropout = 0.5
```

```
In [12]: #Placeholders for Network
         features = tf.placeholder(tf.float32, [None, 32,32,1]) #GrayScale
         y = tf.placeholder(tf.int32, [None])
         labels = tf.one_hot(y, n_classes)
         keep_prob = tf.placeholder(tf.float32)
```

```
In [13]: #Function for Initializing Random TF Variables
         def trun_var(dim=[], mean=mean, std=standard_dev, name=None):
             return tf.Variable(tf.truncated_normal(dim, mean=mean, stddev=std), name=name)
```

```

In [14]: #Initializing Weights and Biases
weights = {
    'conv1': trun_var([5,5,1,6], name='weights_conv1'),
    'conv2': trun_var([5,5,6,16], name='weights_conv2'),
    'full_1': trun_var([400,120], name='weights_full_1'),
    'full_2': trun_var([120,83], name='weights_full_2'),
    'out': trun_var([83, n_classes], name='weights_out')}

biases = {
    'conv1': trun_var([6], name='biases_conv1'),
    'conv2': trun_var([16], name='biases_conv2'),
    'full_1': trun_var([120], name='biases_full_1'),
    'full_2': trun_var([83], name='biases_full_2'),
    'out': trun_var([n_classes], name='biases_out')}

#LeNet CNN Model Implementation

#Convolutional Layer 1
#INPUT 32x32x1
conv1 = tf.nn.conv2d(features, weights['conv1'], strides=[1,1,1,1], padding='VALID')
#28x28x6
conv1 = tf.nn.bias_add(conv1, biases['conv1'])
conv1 = tf.nn.relu(conv1)
conv1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
#14x14x6

#Convolutional Layer 2
conv2_filter = tf.nn.conv2d(conv1, weights['conv2'], strides=[1,1,1,1], padding='VALID')
#10x10x16
conv2_bias = tf.nn.bias_add(conv2_filter, biases['conv2'])
conv2_relu = tf.nn.relu(conv2_bias)
conv2 = tf.nn.max_pool(conv2_relu, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
#5x5x16

#Flatten
conv2 = tf.contrib.layers.flatten(conv2)

#Fully Connected Layer 1
full_1 = tf.add(tf.matmul(conv2, weights['full_1']), biases['full_1'])
#120
full_1 = tf.nn.relu(full_1)
full_1 = tf.nn.dropout(full_1, keep_prob)

#Fully Connected Layer 2
full_2 = tf.add(tf.matmul(full_1, weights['full_2']), biases['full_2'])
#83
full_2 = tf.nn.relu(full_2)
full_2 = tf.nn.dropout(full_2, keep_prob)

#Output Layer
logits = tf.add(tf.matmul(full_2, weights['out']), biases['out'])
#N_CLASSES = 43

```

```

In [15]: ## Training

```

```

#Cost Function
cost = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=labels)

#Adam Optimizer for Back Propagation
optimizer = tf.train.AdamOptimizer(learning_rate=learn_rate).minimize(cost)

```

In [16]: *## Prediction and Accuracy*

```

correct_prediction = tf.equal(tf.argmax(labels, 1), tf.argmax(logits, 1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

```

1.5.3 Train, Validate and Test the Model

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.

In [17]: *### Train your model here.*

```

### Calculate and report the accuracy on the training and validation set.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as well.
### Feel free to use as many code cells as needed.

```

```

init = tf.global_variables_initializer()
save_file = './LeNet_gray_names_15.ckpt' #Weights for GrayScale
# save_file = './LeNet_gray_names_15.ckpt' #Weights with Names for COLOR

```

In [19]: *#Running LeNet and Measuring Accuracy on Validation Set*

```

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(EPOCHS):
        for start_i in range(0,n_train,BATCH_SIZE):
            end_i = start_i + BATCH_SIZE
            sess.run(optimizer, feed_dict={features: X_train_S[start_i:end_i],
                                           y: y_train_S[start_i:end_i] ,
                                           keep_prob: dropout})

            valid_acc = sess.run(accuracy, feed_dict={features: X_valid_gray,
                                                       y: y_valid,
                                                       keep_prob: 1.})

            print('EPOCH: {}'.format(epoch+1))
            print('Validation Accuracy = {}'.format(valid_acc))

    try:
        saver
    except:
        saver = tf.train.Saver()
    saver.save(sess, save_file)

```



```

EPOCH: 1
Validation Accuracy = 0.6517006754875183
EPOCH: 2
Validation Accuracy = 0.817460298538208
EPOCH: 3
Validation Accuracy = 0.8702948093414307
EPOCH: 4
Validation Accuracy = 0.8888888955116272
EPOCH: 5
Validation Accuracy = 0.9077097773551941
EPOCH: 6
Validation Accuracy = 0.915192723274231
EPOCH: 7
Validation Accuracy = 0.9233559966087341
EPOCH: 8
Validation Accuracy = 0.9335600733757019
EPOCH: 9
Validation Accuracy = 0.9285714030265808
EPOCH: 10
Validation Accuracy = 0.9405895471572876
EPOCH: 11
Validation Accuracy = 0.9387755393981934
EPOCH: 12
Validation Accuracy = 0.942630410194397
EPOCH: 13
Validation Accuracy = 0.9487528204917908
EPOCH: 14
Validation Accuracy = 0.9485260844230652
EPOCH: 15
Validation Accuracy = 0.9544217586517334

```

In [18]: *## Test Set Accuracy*

```

with tf.Session() as sess:
    try:
        saver
    except:
        saver = tf.train.Saver()
    saver.restore(sess, save_file)
    total_test_acc = 0
    for start_i in range(0, X_test_gray.shape[0], BATCH_SIZE):
        end_i = start_i + BATCH_SIZE
        X_batch_test, y_batch_test = X_test_gray[start_i:end_i], y_test[start_i:end_i]
        test_acc = sess.run(accuracy, feed_dict={features: X_batch_test,
                                                y: y_batch_test,
                                                keep_prob: 1.})
        total_test_acc += test_acc * X_batch_test.shape[0]

    total_test_acc /= X_test_gray.shape[0]

    print('Test Accuracy = {}'.format(total_test_acc))

Test Accuracy = 0.9351543941860244

```

1.6 Step 3: Test a Model on New Images

To give yourself more insight into how your model is working, download at least five pictures of German traffic signs from the web and use your model to predict the traffic sign type.

You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

1.6.1 Load and Output the Images

```
In [19]: files_num = len(os.listdir('./test-images/'))-1

In [21]: ### Load the images and plot them here.
### Feel free to use as many code cells as needed.
test_imgs = np.zeros([files_num, 32,32,3], dtype=np.uint8)

test_labels = np.array([14, 18, 25, 11, 22, 33, 1, 40, 12, 13, 31, 30, 27])

#Loading and Previewing Images Downloaded from the Internet
for i in range(test_imgs.shape[0]):
    test_imgs[i] = cv2.imread('./test-images/trafficsign{}.jpg'.format(i))
    test_imgs[i] = cv2.cvtColor(test_imgs[i], cv2.COLOR_BGR2RGB)
    test_imgs_gray = normalize(grayScale(test_imgs))
    plt.figure(1, figsize=(20,20))
    for img in range(len(test_imgs)):
        sub_img = plt.subplot(10,5, img+1)
        sub_img.set_title(labels_dict[str(test_labels[img])])
        sub_img.axis('off')
    plt.subplots_adjust(wspace=0.2, hspace=0.5)
    plt.imshow(test_imgs[img])
```



1.6.2 Predict the Sign Type for Each Image

```
In [22]: ### Run the predictions here and use the model
### to output the prediction for each image.
### Make sure to pre-process the images with the
### same pre-processing pipeline used earlier.
### Feel free to use as many code cells as needed.
import pandas as pd
```

```

with tf.Session() as sess:
    saver.restore(sess, save_file)

    results = sess.run(logits, feed_dict={features: test_imgs_gray,
                                           keep_prob: 1.})
    results = np.argmax(results, axis=1)
    result_dict = [str(i) for i in results]
    print('Image\tPrediction\tLabel Predicted')
    print('-'*40)
    for i in range(len(results)):
        if result_dict[i] == str(test_labels[i]):
            prediction = '\u2713'
        else:
            prediction = '\u274C'
        print('{}\t\t\t{}\t\t\t{}'.format(i, results[i], prediction, labels_dict[result_dict[i]]))

```

Image	Prediction	Label Predicted
0	14	✓ Stop
1	18	✓ General caution
2	25	✓ Road work
3	11	✓ Right-of-way at the next intersection
4	22	✓ Bumpy road
5	33	✓ Turn right ahead
6	1	✓ Speed limit (30km/h)
7	40	✓ Roundabout mandatory
8	12	✓ Priority road
9	13	✓ Yield
10	22	X Bumpy road
11	30	✓ Beware of ice/snow
12	27	✓ Pedestrians

1.6.3 Analyze Performance

In [23]: *### Calculate the accuracy for these 5 new images.
 ### For example, if the model predicted 1 out of 5 signs correctly,
 ### it's 20% accurate on these new images.*

```

with tf.Session() as sess:
    saver.restore(sess, save_file)

    test_img_accuracy = sess.run(accuracy, feed_dict={features: test_imgs_gray,
                                                       y: test_labels,
                                                       keep_prob: 1.})
    print('Test Images Accuracy = {:.3f}%'.format(test_img_accuracy * 100))

    Test Images Accuracy = 92.308%

```

1.6.4 Output Top 5 Softmax Probabilities For Each Image Found on the Web

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). `tf.nn.top_k` could prove helpful here.

The example below demonstrates how `tf.nn.top_k` can be used to find the top k predictions for each image.

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if `k=3`, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the corresponding class ids.

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tf.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.07893497,
                0.12789202],
              [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
                0.15899337],
              [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
                0.23892179],
              [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
                0.16505091],
              [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
                0.09155967]])
```

Running it through `sess.run(tf.nn.top_k(tf.constant(a), k=3))` produces:

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
                    [ 0.28086119,  0.27569815,  0.18063401],
                    [ 0.26076848,  0.23892179,  0.23664738],
                    [ 0.29198961,  0.26234032,  0.16505091],
                    [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
                                   [0, 1, 4],
                                   [0, 5, 1],
                                   [1, 3, 5],
                                   [1, 4, 3]], dtype=int32))
```

Looking just at the first row we get `[0.34763842, 0.24879643, 0.12789202]`, you can confirm these are the 3 largest probabilities in `a`. You'll also notice `[3, 0, 5]` are the corresponding indices.

```
In [24]: ### Print out the top five softmax probabilities for the predictions,
        ### on the German traffic sign images found on the web.
        ### Feel free to use as many code cells as needed.
```

```
softmax = tf.nn.softmax(logits)
top_5 = tf.nn.top_k(softmax, k=5)
np.set_printoptions(precision=5)

highest = []
correct = []
for i in result_dict:
    highest.append(labels_dict[i])
for i in test_labels:
    correct.append(labels_dict[str(i)])
```

```

with tf.Session() as sess:
    saver.restore(sess, save_file)
    results, results_index = sess.run(top_5, feed_dict={features: test_imgs_gray,
                                                         keep_prob: 1.})
    sched = pd.DataFrame(results*100, columns=(np.arange(5)+1), index=highest)
    print('Top 5 Probabilites\n{}'.format(results))
    print('\nTop 5 Indices\n{}'.format(results_index))

```

Top 5 Probabilites

```

[[ 9.88516e-01  2.30649e-03  1.88473e-03  1.86310e-03  1.26693e-03]
 [ 9.98160e-01  1.24726e-03  5.86999e-04  5.41087e-06  5.96297e-08]
 [ 9.83891e-01  4.68036e-03  4.48848e-03  3.93070e-03  2.79469e-03]
 [ 9.83671e-01  1.58092e-02  3.91526e-04  9.33847e-05  3.13089e-05]
 [ 9.93305e-01  6.31483e-03  2.10705e-04  9.54587e-05  3.40205e-05]
 [ 9.68048e-01  2.15881e-02  2.19240e-03  1.60021e-03  1.47052e-03]
 [ 9.72511e-01  2.36249e-02  3.50582e-03  2.92685e-04  2.57356e-05]
 [ 8.56425e-01  1.15699e-01  1.30264e-02  7.21028e-03  3.21296e-03]
 [ 9.99984e-01  1.54820e-05  4.09780e-07  3.08537e-08  1.93015e-08]
 [ 1.00000e+00  2.32251e-10  1.38835e-15  3.13608e-16  8.87276e-17]
 [ 6.59242e-01  1.92340e-01  4.92091e-02  4.72197e-02  4.06294e-02]
 [ 9.21510e-01  6.77316e-02  5.68246e-03  4.72241e-03  1.79463e-04]
 [ 8.12275e-01  1.30983e-01  2.63286e-02  1.51958e-02  3.90903e-03]]

```

Top 5 Indices

```

[[14 17 33 15  8]
 [18 26 27 24 25]
 [25 22 31 29 24]
 [11 30 28 21 27]
 [22 29 26 28 17]
 [33 35 39 29  3]
 [ 1  0  4  2  7]
 [40 12 42 32 41]
 [12 40 32 18 41]
 [13 15 12 35  9]
 [22 29 20 30 31]
 [30 11 23 28 21]
 [27 24 18 20 26]]

```

```

In [26]: print('Table Showing Highest Five Percentages / Test Image \
              \n#Labels shown represent the highest prediction')
pd.options.display.float_format = '{:,.3f} %'.format
sched

```

Table Showing Highest Five Percentages / Test Image
#Labels shown represent the highest prediction

```

Out[26]:
      1      2      3      4      5
Stop      98.852 %    0.231 %    0.188 %    0.186 %    0.127 %
General caution      99.816 %    0.125 %    0.059 %    0.001 %    0.000 %
Road work      98.389 %    0.468 %    0.449 %    0.393 %    0.279 %

```

Right-of-way at the next intersection	98.367 %	1.581 %	0.039 %	0.009 %	0.003 %
Bumpy road	99.331 %	0.631 %	0.021 %	0.010 %	0.003 %
Turn right ahead	96.805 %	2.159 %	0.219 %	0.160 %	0.147 %
Speed limit (30km/h)	97.251 %	2.362 %	0.351 %	0.029 %	0.003 %
Roundabout mandatory	85.642 %	11.570 %	1.303 %	0.721 %	0.321 %
Priority road	99.998 %	0.002 %	0.000 %	0.000 %	0.000 %
Yield	100.000 %	0.000 %	0.000 %	0.000 %	0.000 %
Bumpy road	65.924 %	19.234 %	4.921 %	4.722 %	4.063 %
Beware of ice/snow	92.151 %	6.773 %	0.568 %	0.472 %	0.018 %
Pedestrians	81.228 %	13.098 %	2.633 %	1.520 %	0.391 %

1.6.5 Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this [template](#) as a guide. The writeup can be in a markdown or pdf file.

Note: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to ", **"File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

1.7 Step 4 (Optional): Visualize the Neural Network's State with Test Images

This Section is not required to complete but acts as an additional exercise for understanding the output of a neural network's weights. While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what its feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image. From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the [LeNet lab's](#) feature maps looked like for its second convolutional layer you could enter conv2 as the tf_activation variable.

For an example of what feature map outputs look like, check out NVIDIA's results in their paper [End-to-End Deep Learning for Self-Driving Cars](#) in the section Visualization of internal CNN State. NVIDIA was able to show that their network's inner weights had high activations to road boundary lines by comparing feature maps from an image with a clear path to one without. Try experimenting with a similar test to show that your trained network's weights are looking for interesting features, whether it's looking at differences in feature maps from images with or without a sign, or even what feature maps look like in a trained network vs a completely untrained one on the same sign image.

Your output should look something like this (above)

```
In [117]: ### Visualize your network's feature maps here.
          ### Feel free to use as many code cells as needed.

          # image_input: the test image being fed into the network to produce the feature maps
          # tf_activation: should be a tf variable name used during your training procedure
          #               that represents the calculated state of a specific weight layer
```

```

# activation_min/max: can be used to view the activation contrast in more detail,
#     by default matplotlib sets min and max to the actual min and max values of the output
# plt_num: used to plot out multiple different weight feature map sets on the
#     same block, just extend the plt number for each new feature map entry

```

```

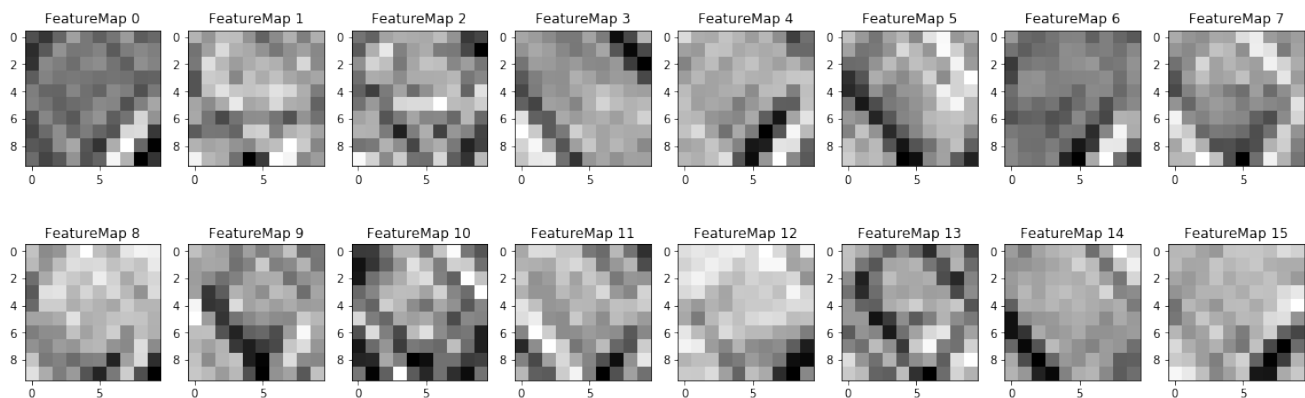
def outputFeatureMap(image_input,
                     tf_activation, activation_min=-1, activation_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network expects
    # with size, normalization, ect if needed
    image_input = normalize(image_input)
    image_input = np.expand_dims(image_input, axis=0)
    # Note: x should be the same name as your network's
    #     tensorflow data placeholder
    # If you get an error tf_activation is not defined it may be having trouble
    # accessing the variable from inside a function
    activation = tf_activation.eval(session=sess,feed_dict={features: image_input})
    featuremaps = activation.shape[3]
    plt.figure(plt_num, figsize=(20,20))
    for featuremap in range(featuremaps):
        # sets the number of feature maps to show on each row and column
        plt.subplot(6,8, featuremap+1)
        plt.title('FeatureMap ' + str(featuremap)) # displays the feature map number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest",
                       vmin=activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,: , featuremap],interpolation="nearest",
                       vmax=activation_max, cmap="gray")
        elif activation_min != -1:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest",
                       vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,: , featuremap], interpolation="nearest",
                       cmap="gray")

```

```

In [136]: with tf.Session() as sess:
            saver.restore(sess, save_file)
            outputFeatureMap(test_imgs_gray[8], conv2_bias)

```



```
In [139]: print('Test Image Resembling the Activations / Feature Map')  
          plt.imshow(test_imgs_gray[8].reshape(32,32), cmap='gray')
```

Test Image Resembling the Activations / Feature Map

```
Out[139]: <matplotlib.image.AxesImage at 0x1129f0e48>
```

