Self-Driving Car Engineer Nanodegree

Deep Learning

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 \n", "**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 (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points (https://review.udacity.com/#!/rubrics/481/view) for this project.

The rubric (https://review.udacity.com/#!/rubrics/481/view) 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.

Step 0: Load The Data

The validation data was already in the zip file, no need to split

In [1]:
```
import cv2
import csv
import time
import glob
import pickle
import numpy as np
import matplotlib.pyplot as plt
from random import randint
from sklearn.utils import shuffle
import tensorflow as tf
from tensorflow.contrib.layers import flatten
```

In [2]:
```
# TODO: Fill this in based on where you saved the training and testing data

training_file = 'traffic-signs-data/train.p'
testing_file = 'traffic-signs-data/test.p'
validation_file= 'traffic-signs-data/valid.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_test, y_test = test['features'], test['labels']
X_valid, y_valid = valid['features'], valid['labels']

print(X_train.shape, y_train.shape, X_valid.shape, y_valid.shape, X_test.shape, y
```

```
(34799, 32, 32, 3) (34799,) (4410, 32, 32, 3) (4410,) (12630, 32, 32, 3) (1263
0,)
```

Step 1: Dataset Summary & Exploration

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

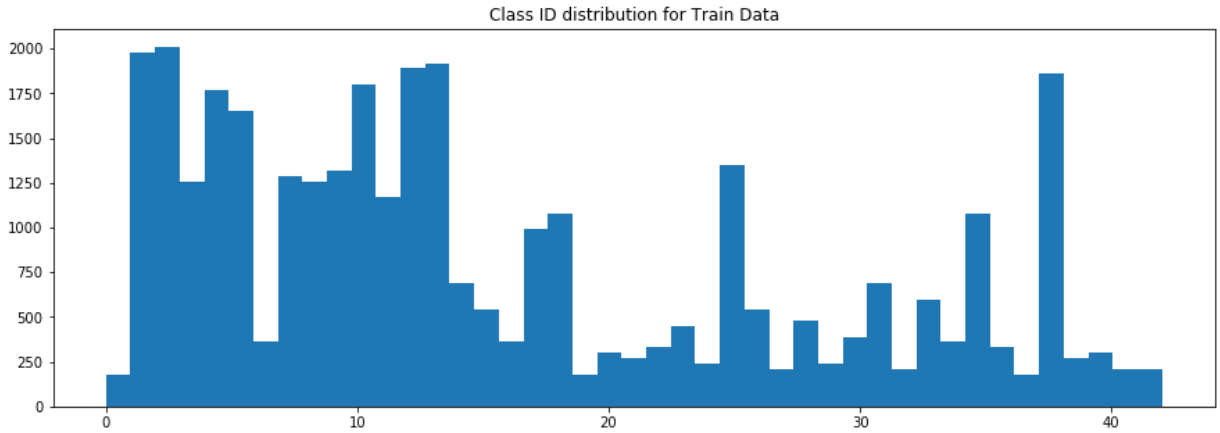
- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing 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**

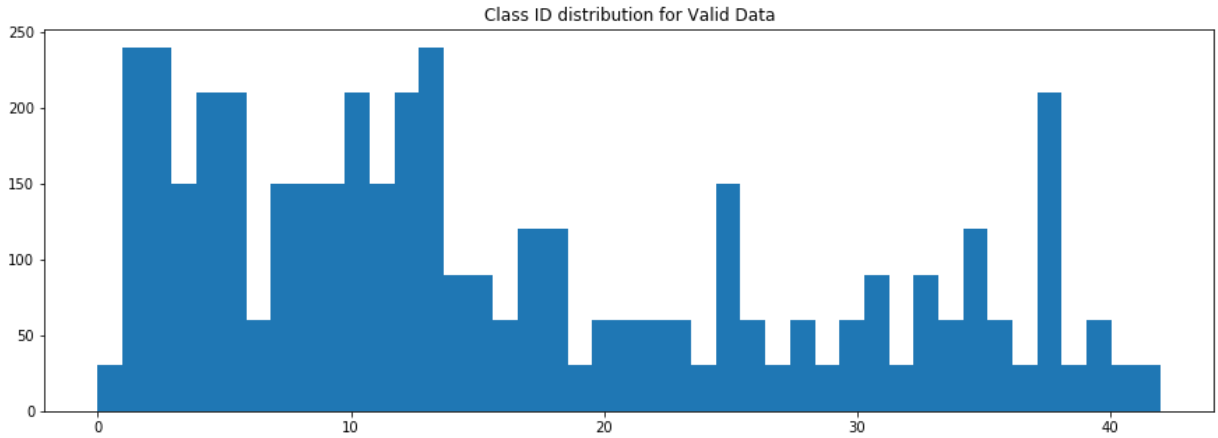
Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the results. For example, the pandas shape method (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html) might be useful for calculating some of the summary results.

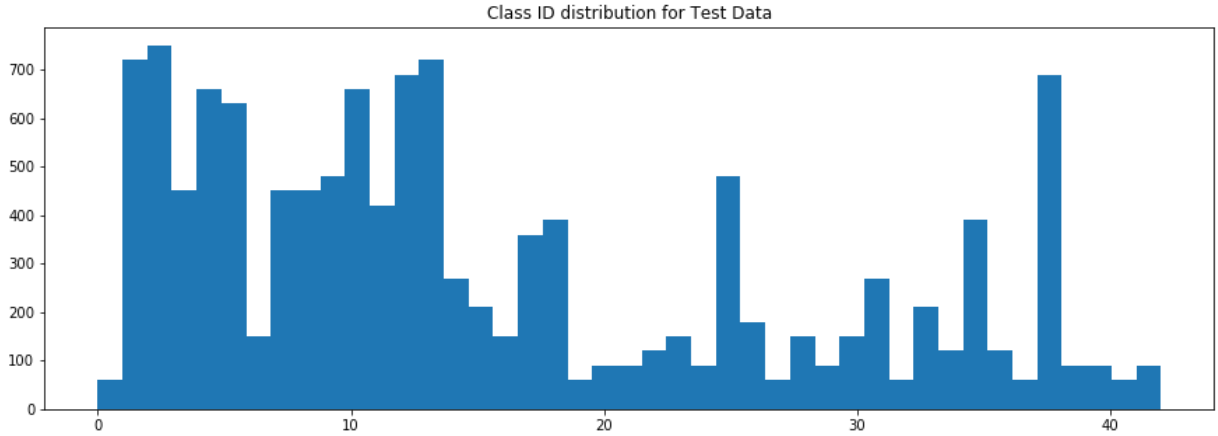
Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

In [3]:
```
### 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 = len(X_train)

# TODO: Number of validation examples
n_validation = len(X_valid)

# TODO: Number of testing examples.
n_test = len(X_test)

# 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 = np.unique(y_train).size

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)


def show_distribution(classIDs, title):
    """
    Plot the traffic sign class distribution
    """
    plt.figure(figsize=(15, 5))
    plt.title('Class ID distribution for {}'.format(title))
    plt.hist(classIDs, bins=n_classes)
    plt.show()


show_distribution(y_train, 'Train Data')
show_distribution(y_valid, 'Valid Data')
show_distribution(y_test,  'Test Data')
```

```
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
```







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 (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) 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 [4]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline

num_of_samples=[]
plt.figure(figsize=(12, 16.5))
for i in range(0, n_classes):
    plt.subplot(10, 5, i+1)
    x_selected = X_train[y_train == i]
    plt.imshow(x_selected[randint(0, 100), :, :, :]) #draw a random image from th
    plt.title(i)
    plt.axis('off')
    num_of_samples.append(len(x_selected))
plt.show()

print("Min number of images per class =", min(num_of_samples))
print("Max number of images per class =", max(num_of_samples))
```


```
Min number of images per class = 180
Max number of images per class = 2010
```

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 (http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

The LeNet-5 implementation shown in the classroom (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) 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 (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). 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.

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, `(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 [ ]:
```
X_valid_original = X_valid
```

In [5]:
```
### Preprocess the data here. It is required to normalize the data. Other preproc
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.

## Normalize the data to zero mean and equal variance

print("Min and Max values original train dataset", np.amin(X_train[0]), np.amax(X

X_train_norm = X_train/127.5-1
X_valid_norm = X_valid/127.5-1
X_test_norm = X_test/127.5-1

print("Min and Max values normalized train dataset", np.amin(X_train_norm[0]), np
```

```
Min and Max values original train dataset 19 113
Min and Max values normalized train dataset -0.850980392157 -0.113725490196
```

In [6]:
```
X_train = X_train_norm
X_valid = X_valid_norm
X_test = X_test_norm
```

In [7]:
```
# # Convert to grayscale
X_train_rgb = X_train
X_train_gray = np.sum(X_train/3, axis=3, keepdims=True)

X_test_rgb = X_test
X_test_gray = np.sum(X_test/3, axis=3, keepdims=True)

X_valid_rgb = X_valid
X_valid_gray = np.sum(X_valid/3, axis=3, keepdims=True)

print(X_train_rgb.shape)
print(X_train_gray.shape)

print(X_test_rgb.shape)
print(X_test_gray.shape)
```

```
(34799, 32, 32, 3)
(34799, 32, 32, 1)
(12630, 32, 32, 3)
(12630, 32, 32, 1)
```

In [8]:
```
X_train = X_train_gray
X_test = X_test_gray
X_valid = X_valid_gray
image_depth_channels = X_train.shape[3]
```

Model Architecture

In [9]:
```
def conv2d(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='VALID')
    x = tf.nn.bias_add(x, b)
#     print(x.shape)
    return tf.nn.relu(x)

def ScottTrafficSignNet(x):
    mu = 0
    sigma = 0.1

    W_one = tf.Variable(tf.truncated_normal(shape=(5, 5, image_depth_channels, 6)
    b_one = tf.Variable(tf.zeros(6))
    layer_one = conv2d(x, W_one, b_one, 1)

    layer_one = tf.nn.max_pool(layer_one, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1
#     print(layer_one.shape)
#     print()

    W_two = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stdde
    b_two = tf.Variable(tf.zeros(16))
    layer_two = conv2d(layer_one, W_two, b_two, 1)

    layer_two = tf.nn.max_pool(layer_two, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1
#     print(layer_two.shape)
#     print()

    W_two_a = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 412), mean = mu, s
    b_two_a = tf.Variable(tf.zeros(412))
    layer_two_a = conv2d(layer_two, W_two_a, b_two_a, 1)

#     print(layer_two_a.shape)
#     print()

    flat = flatten(layer_two_a)

    W_three = tf.Variable(tf.truncated_normal(shape=(412, 122), mean = mu, stddev
    b_three = tf.Variable(tf.zeros(122))
    layer_three = tf.nn.relu(tf.nn.bias_add(tf.matmul(flat, W_three), b_three))
    layer_three = tf.nn.dropout(layer_three, keep_prob)

    W_four = tf.Variable(tf.truncated_normal(shape=(122, 84), mean = mu, stddev =
    b_four = tf.Variable(tf.zeros(84))
    layer_four = tf.nn.relu(tf.nn.bias_add(tf.matmul(layer_three, W_four), b_four
    layer_four = tf.nn.dropout(layer_four, keep_prob)

    W_five = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stddev =
    b_five = tf.Variable(tf.zeros(43))
    layer_five = tf.nn.bias_add(tf.matmul(layer_four, W_five), b_five)

    return layer_five

x = tf.placeholder(tf.float32, (None, 32, 32, image_depth_channels)) #, image_dep
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 43)

keep_prob = tf.placeholder(tf.float32)
```

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.

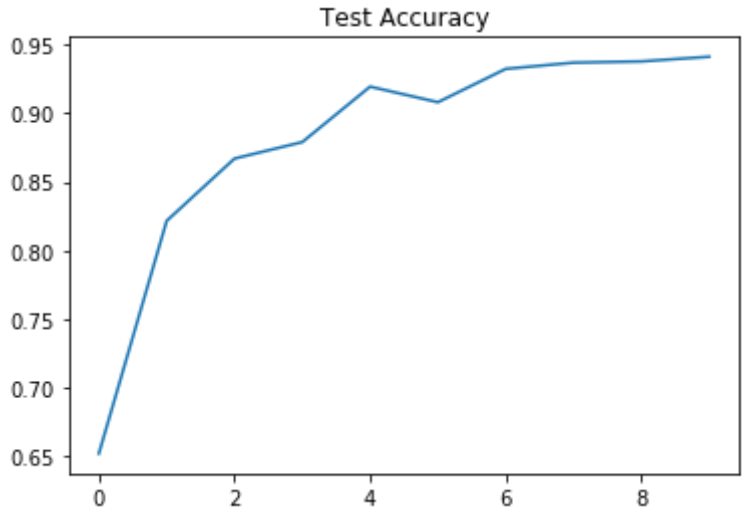
In [10]:
```
## Train your model here.
### Calculate and report the accuracy on the training and validation set.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as well.
### Feel free to use as many code cells as needed.
EPOCHS = 10
BATCH_SIZE = 126

rate = 0.00095

logits = ScottTrafficSignNet(x)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=one
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

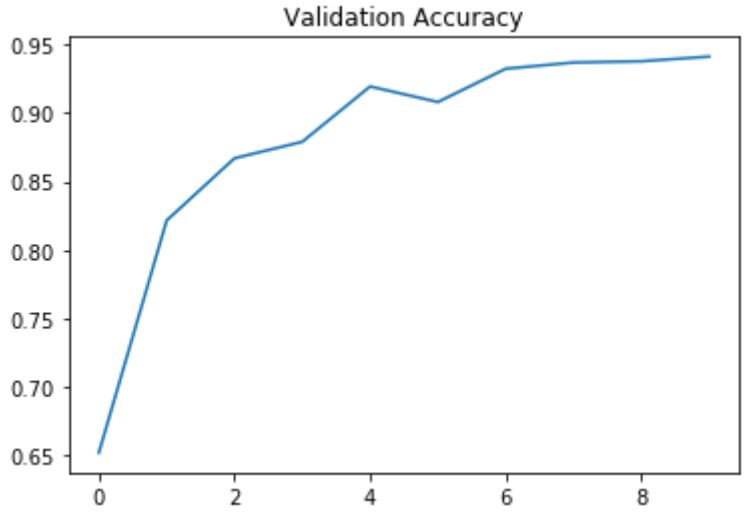
In [11]:
```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    validation_accuracy_figure = []
    test_accuracy_figure = []
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep_

        validation_accuracy = evaluate(X_valid, y_valid)
        validation_accuracy_figure.append(validation_accuracy)

        test_accuracy = evaluate(X_train, y_train)
        test_accuracy_figure.append(test_accuracy)
        print("EPOCH {} ...".format(i+1))
        print("Test Accuracy = {:.3f}".format(test_accuracy))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    saver.save(sess, './ScS_Traffic_Sign_Class')
    print("Model saved")
```

```
Training...

EPOCH 1 ...
Test Accuracy = 0.696
Validation Accuracy = 0.652

EPOCH 2 ...
Test Accuracy = 0.864
Validation Accuracy = 0.822

EPOCH 3 ...
Test Accuracy = 0.930
Validation Accuracy = 0.867

EPOCH 4 ...
Test Accuracy = 0.946
Validation Accuracy = 0.879

EPOCH 5 ...
Test Accuracy = 0.971
Validation Accuracy = 0.919

EPOCH 6 ...
Test Accuracy = 0.969
Validation Accuracy = 0.908

EPOCH 7 ...
Test Accuracy = 0.987
```

```
Validation Accuracy = 0.932

EPOCH 8 ...
Test Accuracy = 0.990
Validation Accuracy = 0.937

EPOCH 9 ...
Test Accuracy = 0.990
Validation Accuracy = 0.938

EPOCH 10 ...
Test Accuracy = 0.994
Validation Accuracy = 0.941

Model saved
```

In [12]:
```
plt.plot(validation_accuracy_figure)
plt.title("Test Accuracy")
plt.show()

plt.plot(validation_accuracy_figure)
plt.title("Validation Accuracy")
plt.show()
```





In [13]:
```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

    train_accuracy = evaluate(X_train, y_train)
    print("Train Accuracy = {:.3f}".format(train_accuracy))

    valid_accuracy = evaluate(X_valid, y_valid)
    print("Valid Accuracy = {:.3f}".format(valid_accuracy))

    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Train Accuracy = 0.994
Valid Accuracy = 0.941
Test Accuracy = 0.923
```

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.

Load and Output the Images

In [14]:
```
def plot_figures(figures, nrows = 1, ncols=1, labels=None):
    fig, axs = plt.subplots(ncols=ncols, nrows=nrows, figsize=(12, 14))
    axs = axs.ravel()
    for index, title in zip(range(len(figures)), figures):
        axs[index].imshow(figures[title], plt.gray())
        if(labels != None):
           axs[index].set_title(labels[index])
        else:
            axs[index].set_title(title)

        axs[index].set_axis_off()

    plt.tight_layout()
```

In [21]:
```
my_images = sorted(glob.glob('./mysigns2/*.jpg'))
my_labels = np.array([11, 25, 28, 3, 14])
name_values = np.genfromtxt('signnames.csv', skip_header=1,
                            dtype=[('myint','i8'), ('mysring','S55')], delimiter=

figures = {}
labels = {}
my_signs_original = []
my_signs_gray = []
my_signs_norm = []
my_signs_gray_norm = []
index = 0
for my_image in my_images:
    img = cv2.cvtColor(cv2.imread(my_image), cv2.COLOR_BGR2RGB)
    print(img.shape)
    img = cv2.resize(img, (32, 32))
    print(img.shape)
    img_gray = np.sum(img/3, axis=2, keepdims=True)
    img_norm = img/127.5-1
    img_gray_norm = img_gray/127.5-1
    my_signs_original.append(img)
    my_signs_gray.append(img_gray)
    my_signs_norm.append(img_norm)
    my_signs_gray_norm.append(img_gray_norm)
    figures[index] = img
    labels[index] = name_values[my_labels[index]][1].decode('ascii')
    index += 1

plot_figures(figures, 3, 2, labels)
```

```
(750, 1000, 3)
(32, 32, 3)
(225, 300, 3)
(32, 32, 3)
(478, 359, 3)
(32, 32, 3)
(345, 359, 3)
(32, 32, 3)
(480, 640, 3)
(32, 32, 3)
```

In [22]:
```
my_signs_original = np.array(my_signs_original)
my_signs = np.array(my_signs_gray_norm)

print(my_signs[0].shape)
```

```
(32, 32, 1)
```

Predict the Sign Type for Each Image

In [23]:
```
### Run the predictions here and use the model to output the prediction for each
### Make sure to pre-process the images with the same pre-processing pipeline use
### Feel free to use as many code cells as needed.

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
#     saver = tf.train.import_meta_graph('./ScS_Traffic_Sign_Class.meta')
    saver.restore(sess, "./ScS_Traffic_Sign_Class")
    my_accuracy = evaluate(my_signs, my_labels)
    print("My Data Set Accuracy = {:.3f}".format(my_accuracy))
```

```
My Data Set Accuracy = 0.800
```

Analyze Performance

In [24]:
```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accu
my_single_item_array = []
my_single_item_label_array = []

for i in range(5):
    my_single_item_array.append(my_signs[i])
    my_single_item_label_array.append(my_labels[i])

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
#         saver = tf.train.import_meta_graph('./ScS_Traffic_Sign_Class.meta')
        saver.restore(sess, "./ScS_Traffic_Sign_Class")
        my_accuracy = evaluate(my_single_item_array, my_single_item_label_array)
        print('Image {}'.format(i+1))
        print("Image Accuracy = {:.3f}".format(my_accuracy))
        print()
```

```
Image 1
Image Accuracy = 1.000

Image 2
Image Accuracy = 1.000

Image 3
Image Accuracy = 0.667

Image 4
Image Accuracy = 0.750

Image 5
Image Accuracy = 0.800
```

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` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#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 correspoding 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. `tk.nn.top_k` is used to choose the three classes with the highest probability:

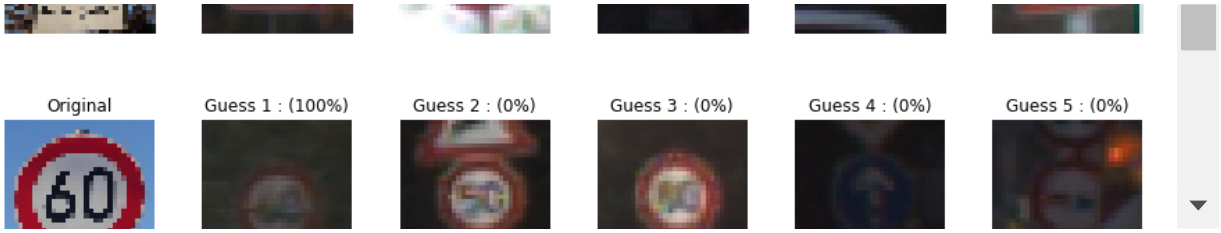
```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.078
93497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 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 [27]:
```
### Print out the top five softmax probabilities for the predictions on the Germa
### Feel free to use as many code cells as needed.

k_size = 5
softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k=k_size)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
#     my_saver = tf.train.import_meta_graph('./ScS_Traffic_Sign_Class.meta')
    saver.restore(sess, "./ScS_Traffic_Sign_Class")
    my_softmax_logits = sess.run(softmax_logits, feed_dict={x: my_signs, keep_pro
    my_top_k = sess.run(top_k, feed_dict={x: my_signs, keep_prob: 1.0})
#     print(my_top_k)

    for i in range(5):
        figures = {}
        labels = {}

        figures[0] = my_signs_original[i]
        labels[0] = "Original"

        for j in range(k_size):
#             print('Guess {} : ({:.0f}%)'.format(j+1, 100*my_top_k[0][i][j]))
            labels[j+1] = 'Guess {} : ({:.0f}%)'.format(j+1, 100*my_top_k[0][i][j
#             figures[j+1] = X_valid[np.argwhere(y_valid == my_top_k[1][i][j])[0]
            figures[j+1] = X_valid_original[np.argwhere(y_valid == my_top_k[1][i]

#         print(len(figures))
#         plot_figures(figures, 1, 6, labels)

        fig, axs = plt.subplots(ncols=6, nrows=1, figsize=(12, 14))
        axs = axs.ravel()
        for index, title in zip(range(len(figures)), figures):
            axs[index].imshow(figures[title])
            if(labels != None):
               axs[index].set_title(labels[index])
            else:
                axs[index].set_title(title)

            axs[index].set_axis_off()

        plt.tight_layout()
```



Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) 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 \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In [ ]: