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

In [1]:
```
import pickle
import cv2
import matplotlib.image as mpimg
import PIL
from PIL import Image
from pylab import *
from sklearn.utils import shuffle
```

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

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_train , Y_train = shuffle(X_train, Y_train)
X_valid, Y_valid = valid['features'], valid['labels']
X_test, Y_test = test['features'], test['labels']
```

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 resu
lts

# TODO: Number of training examples
n_train = X_train.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(open('signnames.csv').readlines())-1

print("Number of training examples =", n_train)
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 testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Include an exploratory visualization of the dataset

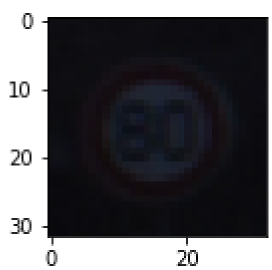
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

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.

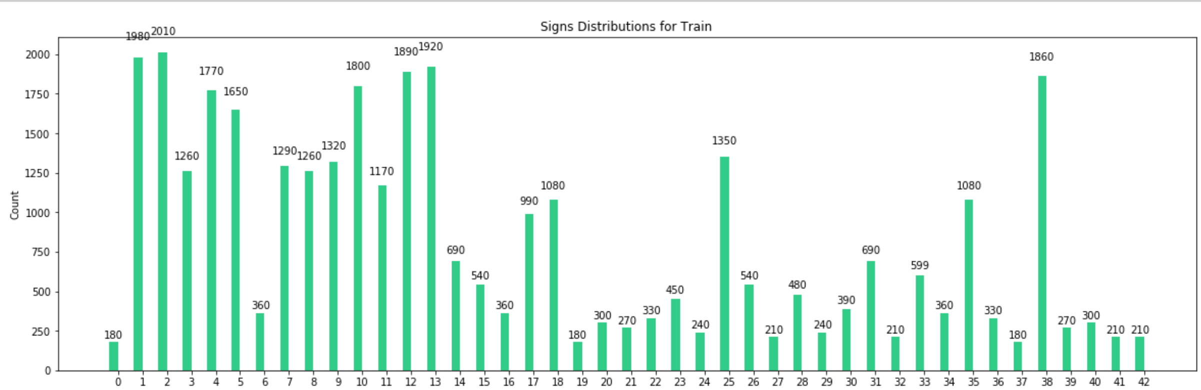
In [4]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import numpy as np
import random
import matplotlib.pyplot as plt

def display_image(image):
    plt.figure(figsize=(2,2))
    plt.imshow(image.squeeze(), cmap="gray")
    plt.show()

# Visualizations will be shown in the notebook.
%matplotlib inline
index = random.randint(0, len(X_train))
image = X_train[index].squeeze()
print('Image label: {}'.format(Y_train[index]))
display_image(image)
```

```
Image label: 5
```



In [5]:

```
def draw_figure(data,labels,color, title):
    N = n_classes
    ind = np.arange(N)  # the x locations for the groups
    width = 0.35       # the width of the bars

    fig, ax = plt.subplots(figsize=(20,6))
    draw_rects = ax.bar(ind, data , width, color=color)

    # add some text for labels, title and axes ticks
    ax.set_ylabel('Count')
    ax.set_title('Signs Distributions for {}'.format(title))
    ax.set_xticks(ind + width / 2)
    ax.set_xticklabels(labels)

    def autolabel(rects):
        for rect in rects:
            height = rect.get_height()
            ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,'%d'
 % int(height), ha='center', va='bottom')

    autolabel(draw_rects)

    plt.show()

#visualizing datasets
train_counts = {}
test_counts = {}
train_labels = [i for i in range(len(Y_train))]
test_labels = [i for i in range(len(Y_test))]
for i in range(n_classes):
    train_counts[i] = 0
    test_counts[i] = 0

for item in Y_train:
    train_counts[item]+=1

_train = [value for key,value in train_counts.items()]
draw_figure(_train, train_labels, '#32CC89', 'Train')
```



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.)

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

In [6]:

```
### Preprocess the data here. Preprocessing steps could include normaliz
ation, converting to grayscale, etc.
### Feel free to use as many code cells as needed.
#converting to grayscale
def convert_to_grayscale(x):
    number_of_images = x.shape[0]
    shape = x[0].shape
    x_gray =np.zeros(shape=(number_of_images,shape[0], shape[1], 1))
    for image_index in range(len(x)):
        x_gray[image_index]= np.reshape(cv2.cvtColor(x[image_index],
cv2.COLOR_RGB2GRAY), (32,32,1))
    return x_gray

def normalize(images):
    normalized_images = np.zeros_like(images)
    for image_index in range(len(images)):
        normalized_images[image_index] = (images[image_index]-128)/128
    return normalized_images


def preprocess(images):
    grayscaled_images = convert_to_grayscale(images)
    normalized = normalize(grayscaled_images)
    return normalized

X_train_normalized = preprocess(X_train)
```

In [7]:

```
print('Training size before adding noise: {} {}'.format(len(X_train_norm
alized),len(Y_train)))
# add noise
for i in range(500):
    index = random.randint(0, len(X_train_normalized))
    image = X_train_normalized[index]
    image_label = Y_train[index]
    noise = np.empty(image.shape, np.float32)
    noise = cv2.randn(noise,(-.04),(0.04))
    image_copy = np.copy(image) + noise
    X_train_normalized = np.concatenate((X_train_normalized,
np.zeros([1,32,32,1])))
    X_train_normalized[len(X_train_normalized)-1] = image_copy
    Y_train = np.concatenate((Y_train, [image_label]))

print('Training size after adding noise: {} {}'.format(len(X_train_norma
lized),len(Y_train)))
#shuffle
X_train_normalized , Y_train = shuffle(X_train_normalized, Y_train)
```

```
Training size before adding noise: 34799 34799
Training size after adding noise: 35299 35299
```

Model Architecture

In [8]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.
```

```
import tensorflow as tf
from tensorflow.contrib.layers import flatten

filter_depth_c1 = 6
filter_depth_c2 = 16
mu = 0
sigma = 0.1

weights = {
    'c1':tf.Variable(tf.truncated_normal(shape=(5, 5, 1,
filter_depth_c1), mean = mu, stddev = sigma), name='weights_c1'),
    'c2':tf.Variable(tf.truncated_normal(shape=(5,5,6, filter_depth_c2),
 mean = mu, stddev = sigma), name='weights_c2'),
    'f1':tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, st
ddev = sigma), name="weights_f1"),
    'f2':tf.Variable(tf.truncated_normal(shape=(120,84), mean = mu, stdd
ev = sigma), name="weights_f2"),
    'output':tf.Variable(tf.truncated_normal(shape=(84,n_classes)))
}
biases = {
    'c1':tf.Variable(tf.zeros(filter_depth_c1), name="bias_c1"),
    'c2':tf.Variable(tf.zeros(filter_depth_c2), name="bias_c2"),
    'f1':tf.Variable(tf.zeros(120),name="bias_f1"),
    'f2':tf.Variable(tf.zeros(84), name="bias_f2"),
    'output':tf.Variable(tf.zeros(n_classes))
}

def conv2d(input,weights, bias):
    c = tf.nn.conv2d(input, weights, strides=[1,1,1,1], padding='VALID')
    c = tf.nn.bias_add(c,bias)
    c = tf.nn.relu(c)
    c = tf.nn.max_pool(c, ksize=[1,2,2,1], strides=[1,2,2,1], padding='V
ALID')
    return c

def fully_connected(x, weights, bias):
    f = tf.add(tf.matmul(x, weights), bias)
    f = tf.nn.relu(f)
    f = tf.nn.dropout(f, drop_out)
    return f

def model(x):

    #convnet1 32x32x1 -> 14x14x6
    conv1 = conv2d(x, weights['c1'], biases['c1'])

    #convnet2 14x14x6 -> 5x5x16
    conv2 = conv2d(conv1, weights['c2'], biases['c2'])

    #fully_connected 120 nodes
    conv2_flat = flatten(conv2)
    f1 = fully_connected(conv2_flat, weights['f1'], biases['f1'])

    # fully_connected 84 nodes
    f2 = fully_connected(f1, weights['f2'], biases['f2'])

    #output 10 nodes
```

```
    logits = tf.add(tf.matmul(f2,weights['output']), biases['output'])

    return (conv1,conv2, f1, f2, logits)
```

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 [9]:
```
### Train your model here.
### Calculate and report the accuracy on the training and validation se
t.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as we
ll.
### Feel free to use as many code cells as needed.
EPOCH = 30
BATCH_SIZE= 128
x = tf.placeholder(tf.float32, (None, 32,32,1), name="x")
y = tf.placeholder(tf.int32, (None), name="y")
drop_out = tf.placeholder(tf.float32, name="drop_out")
one_hot_y = tf.one_hot(y, n_classes)
learning_rate = 0.001
conv1, conv2, f1,f2 , logits = model(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_
y)
cost = tf.reduce_mean(cross_entropy)
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
```

In [10]:
```
#model validation

is_correct = tf.equal(tf.argmax(logits,1), tf.argmax(one_hot_y,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
def evaluate(x_data, y_data):
    size = len(x_data)
    overall_accuracy = 0
    session = tf.get_default_session()
    for batch_number in range(0, size, BATCH_SIZE):
        start = batch_number
        end = batch_number + BATCH_SIZE
        batch_x = x_data[start:end]
        batch_y = y_data[start:end]
        results = session.run(accuracy, feed_dict = {x:batch_x, y:batch_
y, drop_out: 1})
        overall_accuracy += (results* len(batch_x))
    return overall_accuracy/size
```

In [19]:
```
saver = tf.train.Saver()
init = tf.global_variables_initializer()
x_train_size = len(X_train_normalized)
with tf.Session() as session:
    session.run(init)
    for epoch in range(EPOCH):

        x_train, y_train= shuffle(X_train_normalized, Y_train)
        for batch_number in range(0, x_train_size, BATCH_SIZE):
            end = batch_number+BATCH_SIZE
            start = batch_number
            batch_x = x_train[start:end]
            batch_y = y_train[start:end]
            feed_dict = {x:batch_x, y: batch_y, drop_out:0.5}
            session.run(optimizer, feed_dict=feed_dict)

        x_valid_normalized = preprocess(X_valid)
        accuracy_validation = evaluate(x_valid_normalized, Y_valid)
        print('EPOCH : {}'.format(epoch))
        print('accuracy: {:.3f}'.format(accuracy_validation))

    saver.save(session,'./traffic_sign_model')
    print('model saved')
```

```
EPOCH : 0
accuracy: 0.523
EPOCH : 1
accuracy: 0.757
EPOCH : 2
accuracy: 0.834
EPOCH : 3
accuracy: 0.856
EPOCH : 4
accuracy: 0.873
EPOCH : 5
accuracy: 0.889
EPOCH : 6
accuracy: 0.897
EPOCH : 7
accuracy: 0.918
EPOCH : 8
accuracy: 0.919
EPOCH : 9
accuracy: 0.924
EPOCH : 10
accuracy: 0.927
EPOCH : 11
accuracy: 0.932
EPOCH : 12
accuracy: 0.937
EPOCH : 13
accuracy: 0.931
EPOCH : 14
accuracy: 0.942
EPOCH : 15
accuracy: 0.946
EPOCH : 16
accuracy: 0.936
EPOCH : 17
accuracy: 0.942
EPOCH : 18
accuracy: 0.948
EPOCH : 19
accuracy: 0.944
EPOCH : 20
accuracy: 0.947
EPOCH : 21
accuracy: 0.953
EPOCH : 22
accuracy: 0.948
EPOCH : 23
accuracy: 0.953
EPOCH : 24
accuracy: 0.947
EPOCH : 25
accuracy: 0.940
EPOCH : 26
accuracy: 0.951
EPOCH : 27
accuracy: 0.945
EPOCH : 28
```

```
accuracy: 0.953
EPOCH : 29
accuracy: 0.951
model saved
```

Step 3: Test a Model on New Images

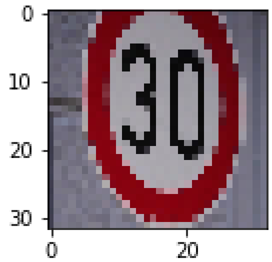
To give yourself more insight into how your model is working, download at least five pictures of German traffic signs from the web and use your model to predict the traffic sign type.

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

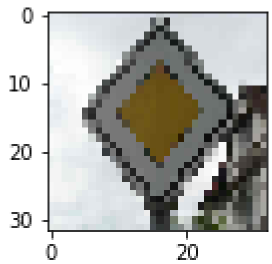
Load and Output the Images

In [20]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
saver = tf.train.Saver()
#collecting 5 images from web
x_web = np.zeros_like(X_train[:7,:,:,:])
x_web[0] = np.asarray(Image.open('sample_traffic_signs/1.jpg').resize((3
2,32)))
x_web[1] = np.asarray(Image.open('sample_traffic_signs/2.jpg').resize((3
2,32)))
x_web[2] = np.asarray(Image.open('sample_traffic_signs/3.jpg').resize((3
2,32)))
x_web[3] = np.asarray(Image.open('sample_traffic_signs/4.jpg').resize((3
2,32)))
x_web[4] = np.asarray(Image.open('sample_traffic_signs/5.jpg').resize((3
2,32)))
x_web[5] = np.asarray(Image.open('sample_traffic_signs/6.jpg').resize((3
2,32)))
x_web[6] = np.asarray(Image.open('sample_traffic_signs/7.jpg').resize((3
2,32)))

y_web = np.array([1, 12, 14, 25 , 31,23, 21], dtype=np.uint8)

with tf.Session() as session:
    saver.restore(session, tf.train.latest_checkpoint('.'))
    for image_index in range(len(x_web)):
        image = x_web[image_index].squeeze()
        print('Image #{}'.format(image_index))
        display_image(image)
```

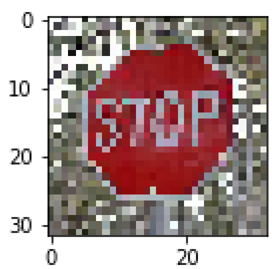
```
Image #0
```



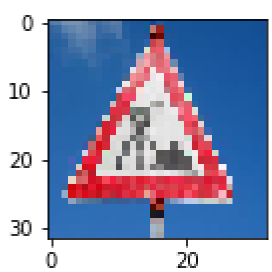

```
Image #1
```



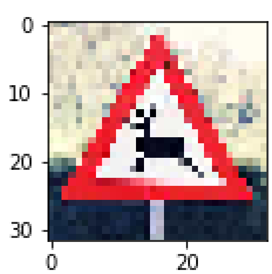

```
Image #2
```




```
Image #3
```




```
Image #4
```




```
Image #5
```

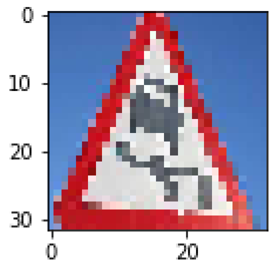
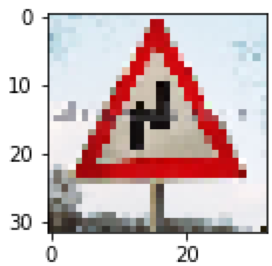


Image #6



Predict the Sign Type for Each Image

In [21]:
```
### 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 pip
eline used earlier.
### Feel free to use as many code cells as needed.

#pre-processing
x_web_normalized = preprocess(x_web)
overall_accuracy = 0
with tf.Session() as session:
    saver.restore(session, tf.train.latest_checkpoint('.'))
    for image_index in range(len(x_web)):
        overall_accuracy = evaluate(x_web_normalized, y_web)
```

Analyze Performance

In [22]:
```
### 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.
print('Overall Accuracy : {:.3f}'.format(overall_accuracy))
print('=====================================')
```

```
Overall Accuracy : 0.571
=====================================
```

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)](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,  0.078934
97,
         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:

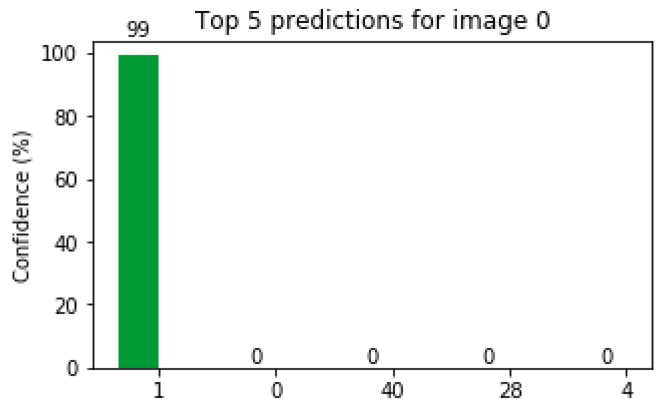
```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],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

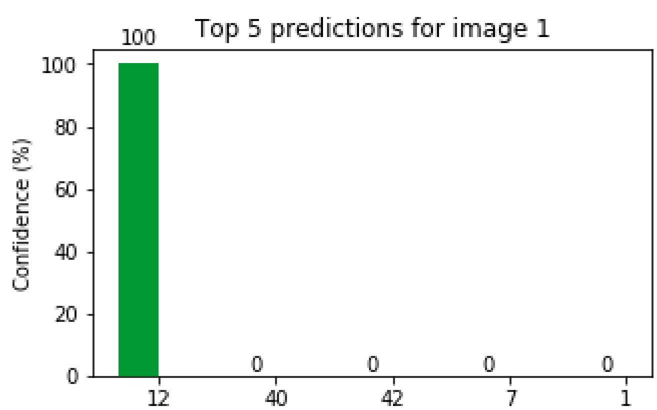
In [23]:
```
def draw_top_k(labels,data,image_index, k_size, color):
    N = k_size
    ind = np.arange(N)  # the x locations for the groups
    width = 0.35
    data*=100 # the width of the bars
    data = data.astype(int)
    fig, ax = plt.subplots(figsize=(5,3))
    draw_rects = ax.bar(ind, data , width, color=[color, '#808080', '#80
8080'])

    # add some text for labels, title and axes ticks
    ax.set_ylabel('Confidence (%)')
    ax.set_title('Top 5 predictions for image {}'.format(image_index))
    ax.set_xticks(ind + width / 2)
    ax.set_xticklabels(labels)

    def autolabel(rects):
        for rect in rects:
            height = rect.get_height()
            ax.text(rect.get_x() + rect.get_width()/2., 1.05*height,'%d'
 % int(height), ha='center', va='bottom')

    autolabel(draw_rects)
    plt.show()
```

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)
predictions = tf.argmax(logits,1)
k_size=5
top_k= tf.nn.top_k(softmax, k=k_size)
signs = list(open('signnames.csv').readlines())
with tf.Session() as session:
    saver.restore(session, tf.train.latest_checkpoint('.'))
    y_top_k = session.run(top_k, feed_dict={x:x_web_normalized,
drop_out:1})
    y_is_correct = session.run(is_correct, feed_dict=
{x:x_web_normalized,y:y_web, drop_out:1} )
    for result_index in range(len(x_web_normalized)):
        prediction_result= 'Incorrect'
        color = '#ff0000'
        if y_is_correct[result_index] == True:
            prediction_result = 'Correct'
            color='#009933'
        print('Image #{} [Predicted: {}]'.format(result_index,prediction
_result ))
        p_values = y_top_k.values[result_index]
        indices = y_top_k.indices[result_index]
        print('Predicted Sign:
{}'.format(signs[indices[0]+1].rstrip('\n')))
        print('Actual Sign: {}'.format(signs[y_web[result_index]+1].rstr
ip('\n')))

        draw_top_k(indices, p_values, result_index, k_size, color)
        print("Top {} results:".format(k_size))
        for index in range(k_size):
            print('probablity: {} , sign: {}'.format(p_values[index],sig
ns[indices[index]+1].rstrip('\n')))

        print('==================================================\n')
```

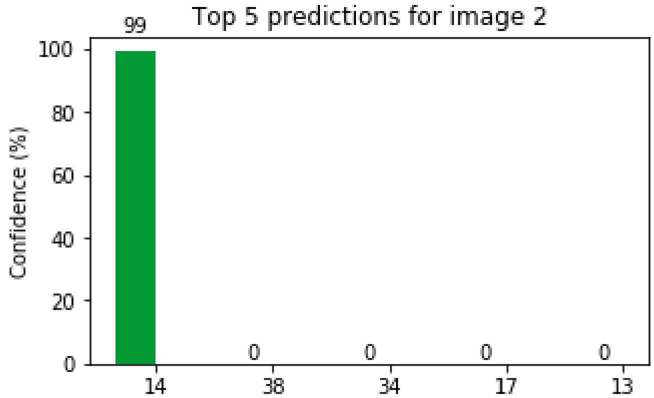
```
Image #0 [Predicted: Correct]
Predicted Sign: 1,Speed limit (30km/h)
Actual Sign: 1,Speed limit (30km/h)
```



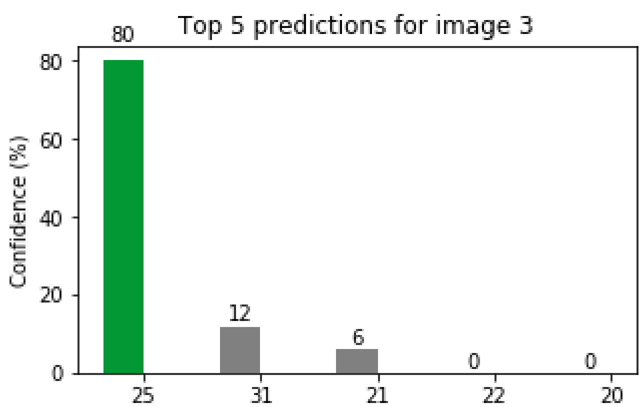

```
Top 5 results:
probablity: 99.619384765625 , sign: 1,Speed limit (30km/h)
probablity: 0.30610108375549316 , sign: 0,Speed limit (20km/h)
probablity: 0.038351334631443024 , sign: 40,Roundabout mandatory
probablity: 0.011021287180483341 , sign: 28,Children crossing
probablity: 0.00678263371810317 , sign: 4,Speed limit (70km/h)
==================================================

Image #1 [Predicted: Correct]
Predicted Sign: 12,Priority road
Actual Sign: 12,Priority road
```



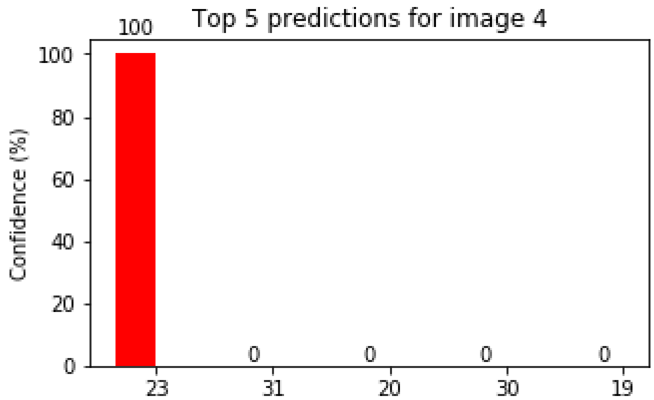

```
Top 5 results:
probablity: 100.0 , sign: 12,Priority road
probablity: 4.700046076777653e-07 , sign: 40,Roundabout mandatory
probablity: 1.1679628618421756e-11 , sign: 42,End of no passing by vehi
cles over 3.5 metric tons
probablity: 4.077430260833806e-15 , sign: 7,Speed limit (100km/h)
probablity: 4.875464999067405e-16 , sign: 1,Speed limit (30km/h)
==================================================

Image #2 [Predicted: Correct]
Predicted Sign: 14,Stop
Actual Sign: 14,Stop
```



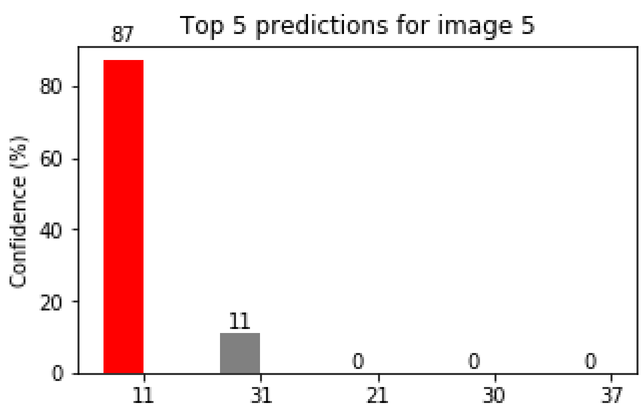

```
Top 5 results:
probablity: 99.99876403808594 , sign: 14,Stop
probablity: 0.001188850961625576 , sign: 38,Keep right
probablity: 5.336221875040792e-05 , sign: 34,Turn left ahead
probablity: 6.982488116591412e-07 , sign: 17,No entry
probablity: 4.4473472371464595e-07 , sign: 13,Yield
==================================================

Image #3 [Predicted: Correct]
Predicted Sign: 25,Road work
Actual Sign: 25,Road work
```



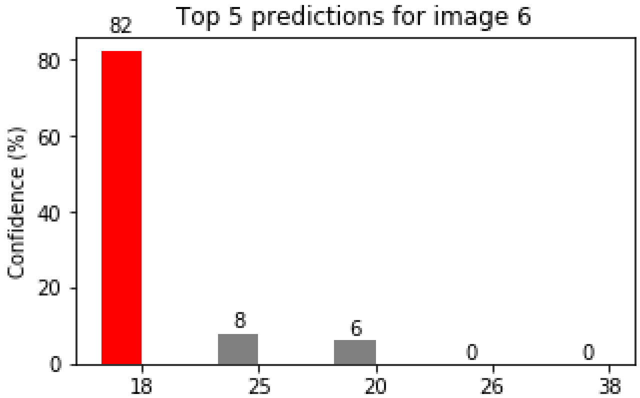

```
Top 5 results:
probablity: 80.46527862548828 , sign: 25,Road work
probablity: 12.754717826843262 , sign: 31,Wild animals crossing
probablity: 6.550268173217773 , sign: 21,Double curve
probablity: 0.13391906023025513 , sign: 22,Bumpy road
probablity: 0.04608609527349472 , sign: 20,Dangerous curve to the right
==================================================

Image #4 [Predicted: Incorrect]
Predicted Sign: 23,Slippery road
Actual Sign: 31,Wild animals crossing
```




```
Top 5 results:
probablity: 100.0 , sign: 23,Slippery road
probablity: 9.123946256295312e-07 , sign: 31,Wild animals crossing
probablity: 8.476145012537017e-07 , sign: 20,Dangerous curve to the rig
ht
probablity: 1.1697153468048782e-07 , sign: 30,Beware of ice/snow
probablity: 3.1184420379304356e-08 , sign: 19,Dangerous curve to the le
ft
==================================================

Image #5 [Predicted: Incorrect]
Predicted Sign: 11,Right-of-way at the next intersection
Actual Sign: 23,Slippery road
```



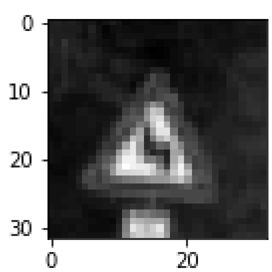

```
Top 5 results:
probablity: 87.93341064453125 , sign: 11,Right-of-way at the next inter
section
probablity: 11.897724151611328 , sign: 31,Wild animals crossing
probablity: 0.08555130660533905 , sign: 21,Double curve
probablity: 0.04106033593416214 , sign: 30,Beware of ice/snow
probablity: 0.019457001239061356 , sign: 37,Go straight or left
==================================================

Image #6 [Predicted: Incorrect]
Predicted Sign: 18,General caution
Actual Sign: 21,Double curve
```



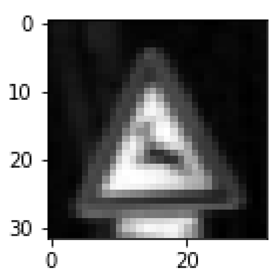

```
Top 5 results:
probablity: 82.82020568847656 , sign: 18,General caution
probablity: 8.873628616333008 , sign: 25,Road work
probablity: 6.7343974113464355 , sign: 20,Dangerous curve to the right
probablity: 0.5950281023979187 , sign: 26,Traffic signals
probablity: 0.4056157171726227 , sign: 38,Keep right
==================================================
```

In [25]:
```
def display_train_data(index, label):
    image = X_train_normalized[index].squeeze()
    print('label {}'.format(label))
    plt.figure(figsize=(2,2))
    plt.imshow(image, cmap='gray')
    plt.show()

#printing a random sample image from training data set with failed image
s signs
# 23 , 21 , 25
random_sample = random.randint(0, 20)
index_31 = np.where(Y_train==31)[0][random_sample]
index_21 = np.where(Y_train==21)[0][random_sample]
index_23 = np.where(Y_train==23)[0][random_sample]

print('Sample Images from training set')
display_train_data(index_21, signs[22])
display_train_data(index_31, signs[32])
display_train_data(index_23, signs[24])
```

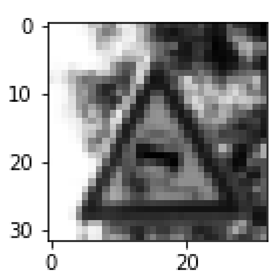
```
Sample Images from training set
label 21,Double curve
```



```
label 31,Wild animals crossing
```



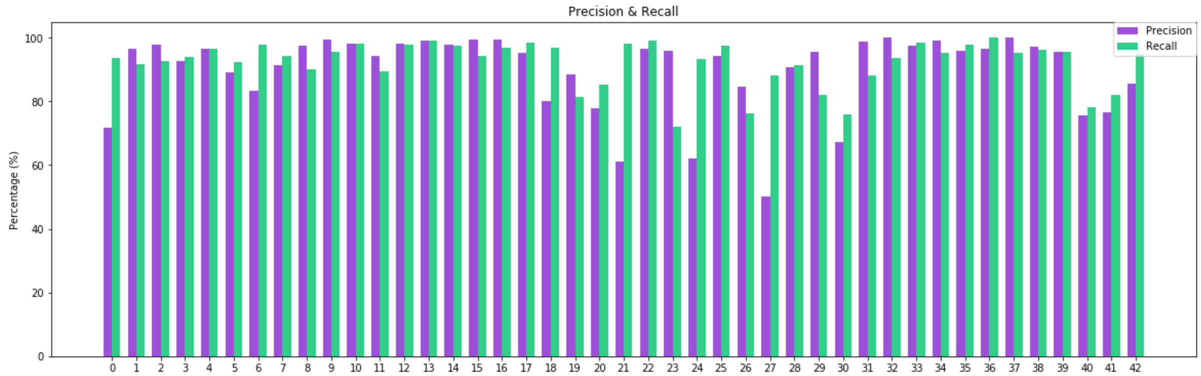
```
label 23,Slippery road
```



In [26]:
```
#Running Test set

with tf.Session() as session:
    saver.restore(session, tf.train.latest_checkpoint('.'))
    x_test_normalized = preprocess(X_test)
    overall_accuracy = evaluate(x_test_normalized, Y_test)
    y_results = session.run(predictions, feed_dict={x:x_test_normalized,
 y:Y_test, drop_out:1})
    print('Overall Accuracy {:.3f}'.format(overall_accuracy))
```

```
Overall Accuracy 0.938
```

In [27]:
```
#calculating precision & recall
test_performance = {}
performance_analysis = {}
s = []
r = []
p = []
signs = list(open('signnames.csv').readlines())[1:]
for sign in signs:
    sign_number = int(sign.split(',')[0])
    sing_name = sign.rstrip('\n')
    test_performance[sign_number]={'TP':0, 'FP':0, 'FN':0, 'sign':sing_n
ame}
    performance_analysis[sign_number] = {'recall': 0. ,'precision':0.,
'sign': sing_name}
    s.append(sign_number)

#get the predictions
with tf.Session() as session:
    saver.restore(session,tf.train.latest_checkpoint('.'))
    test_predictions = session.run(predictions, feed_dict={x:x_test_norm
alized,y:Y_test, drop_out:1})
    for image_index in range(len(Y_test)):
        if test_predictions[image_index] == Y_test[image_index]:
            test_performance[Y_test[image_index]]['TP']+=1
        else:
            test_performance[Y_test[image_index]]['FP']+=1
            test_performance[test_predictions[image_index]]['FN']+=1

    for key,value in test_performance.items():
        performance_analysis[key]['recall'] = value['TP']/ (value['TP']+
 value['FN'])
        performance_analysis[key]['precision'] = value['TP']/ (value['T
P']+ value['FP'])
        r.append(100*value['TP']/ (value['TP']+ value['FN']))
        p.append(100*value['TP']/ (value['TP']+ value['FP']))



N = 43
ind = np.arange(N)  # the x locations for the groups
width = 0.35       # the width of the bars

fig, ax = plt.subplots(figsize=(20,6))
precision_rects = ax.bar(ind, p, width, color='#9D51D8')
recall_rects = ax.bar(ind + width, r, width, color='#32CC89')

# add some text for labels, title and axes ticks
ax.set_ylabel('Percentage (%)')
ax.set_title('Precision & Recall')
ax.set_xticks(ind + width / 2)
ax.set_xticklabels(s)

ax.legend((precision_rects[0], recall_rects[0]), ('Precision',
'Recall'), loc=1, borderaxespad=0.)

plt.show()
```

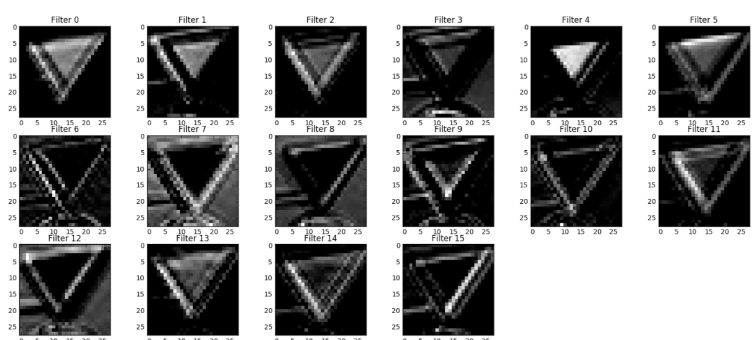

Precision & Recall
Precision
Recall
Percentage (%)
0
20
40
60
80
100
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42

Step 4: Visualize the Neural Network's State with Test Images

This Section is not required to complete but acts as an additional excersise for understaning 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 it's 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 (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) feature maps looked like for it's second convolutional layer you could enter conv2 as the tf_activation variable.

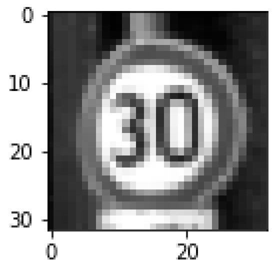
For an example of what feature map outputs look like, check out NVIDIA's results in their paper End-to-End Deep Learning for Self-Driving Cars (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/) in the section Visualization of internal CNN State. NVIDIA was able to show that 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.



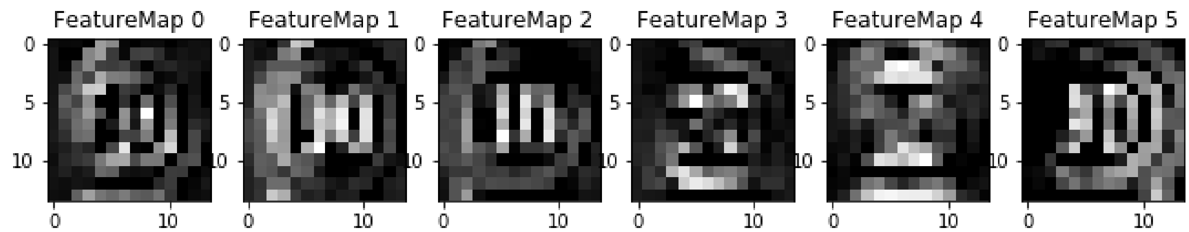
Your output should look something like this (above)

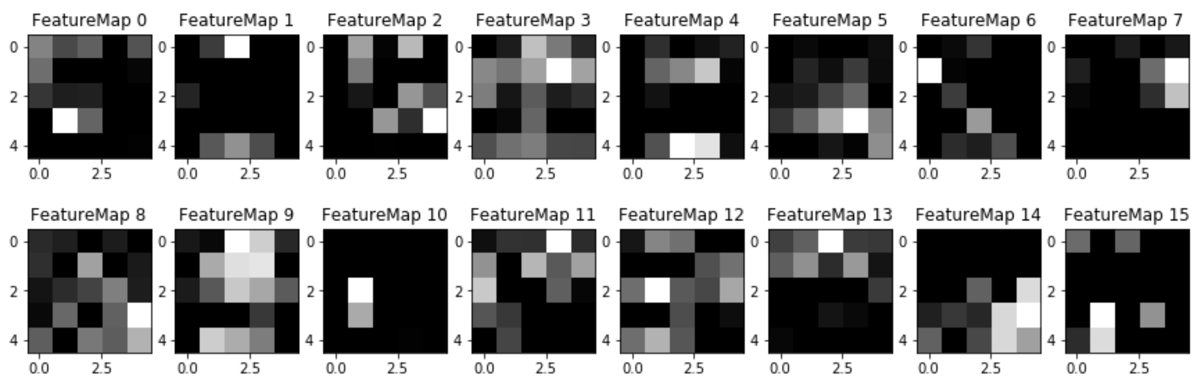
In [32]:
```
### Visualize your network's feature maps here.
### Feel free to use as many code cells as needed.

# 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 lay
```

```
er
# activation_min/max: can be used to view the activation contrast in mor
e detail, by default matplot 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 o
n the same block, just extend the plt number for each new feature map en
try

def outputFeatureMap(image_input, tf_activation, sess,image_index, activ
ation_min=-1, activation_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your networ
k expects
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data
 placeholder variable
    # If you get an error tf_activation is not defined it maybe having t
rouble accessing the variable from inside a function
    activation = tf_activation.eval(session=sess,feed_dict={x : image_in
put, drop_out:0.5})
    featuremaps = activation.shape[3]
    plt.figure(plt_num, figsize=(15,15))
    for featuremap in range(featuremaps):
        plt.subplot(6,8, featuremap+1) # sets the number of feature maps
 to show on each row and column
        plt.title('FeatureMap ' + str(featuremap)) # displays the featur
e map number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[image_index,:,:, featuremap], interpol
ation="nearest", vmin =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[image_index,:,:, featuremap], interpol
ation="nearest", vmax=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[image_index,:,:, featuremap], interpol
ation="nearest", vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[image_index,:,:, featuremap], interpol
ation="nearest", cmap="gray")

with tf.Session() as session:
    image_index = 1 #60#12
    image = x_test_normalized[image_index].squeeze()
    plt.figure(figsize=(2,2))
    plt.imshow(image, cmap="gray")
    plt.show()
    print('Prediction: {}'.format(signs[y_results[image_index]]))
    saver.restore(session, tf.train.latest_checkpoint('.'))
    outputFeatureMap(x_test_normalized, conv1 , sess=session, image_inde
x=image_index)
    outputFeatureMap(x_test_normalized, conv2 , sess=session, plt_num=6,
 image_index=image_index)
```



```
Prediction: 1,Speed limit (30km/h)
```





Question 9

Discuss how you used the visual output of your trained network's feature maps to show that it had learned to look for interesting characteristics in traffic sign images

Anwser: Feature map is the result of sample image sign (1,Speed limit 30km/h) through conv1 [Feature-maps 0-5] and conv2 [Feature-maps 0-15] layers.
One of the most interesting charactersitics in these feature maps is the figure-contours detected by network. conv1 layer is focusing on the shape of 30 and the circle around the sign. Output of conv1 is max-pooled to a smaller pixel scale there are less pixels in conv2, so looks like its feature maps are more detailing out colors and orientations/locations of the colors.

Answer: