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 [250]:

```
# Load pickled data
import tensorflow as tf
import pickle
import os

# TODO: Fill this in based on where you saved the training and testing data
# 打开目录为'./traffic-signs-data/train.p',
training_file = os.path.join('traffic-signs-data','train.p')
testing_file = os.path.join('traffic-signs-data','test.p')
validation_file = os.path.join('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_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 [251]:

```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
import numpy as np

# Split 20% of the training data for validation
from sklearn.model_selection import train_test_split

X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_siz
e = 0.2, random_state = 0)

assert(len(X_train) == len(y_train))
assert(len(X_valid) == len(y_valid))
assert(len(X_test) == len(y_test))

# 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 = len(set(y_train))
#或者：n_classes = len(set(train['labels']))


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 = 27839
Number of validation examples = 6960
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?

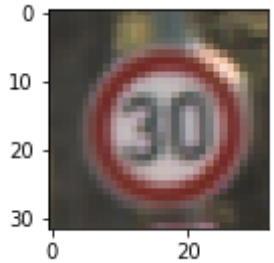
In [253]:

```
# Creat a dictionary "signname_dict" {'index' : 'sign name'}
import csv

with open('signnames.csv', mode='r') as infile:
    reader = csv.reader(infile)
    with open('singnames_new.csv', mode='w') as outfile:
        writer = csv.writer(outfile)
        signname_dict = {rows[0]:rows[1] for rows in reader}
#print(signname_dict['1'])  #As a test
```

In [254]:

```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline


### Show a random image in X_train and print its data

index = random.randint(0, len(X_train))
image = X_train[index].squeeze()

plt.figure(figsize=(2,2))
plt.imshow(image)
print('Image Index ID: {}'.format(index))
print('Class ID: {}'.format(y_train[index]))
print('Sign Name: {}'.format(signname_dict[str(y_train[index])]))
print('The shape of the image is {}'.format(image.shape))
```

```
Image Index ID: 24530
Class ID: 1
Sign Name: Speed limit (30km/h)
The shape of the image is (32, 32, 3)
```



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 [255]:

```
### Preprocess the data here. It is required to normalize the data. Other prepro
cessing steps could include
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.
```

In [256]:

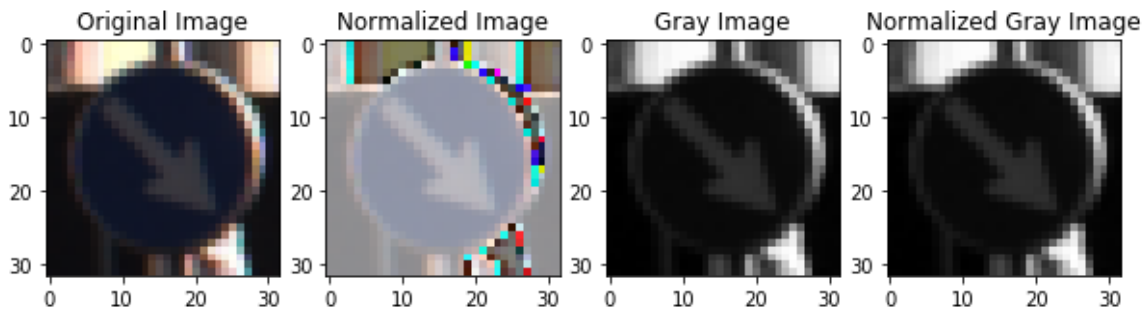
```
# This block is used to shuffle the data for training, validation and test.
# Shuffling is alwasy done first.
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
X_valid, y_valid = shuffle(X_valid, y_valid)
X_test, y_test = shuffle(X_test, y_test)
print('Train/talidation/test are now shuffled.')
```

```
Train/talidation/test are now shuffled.
```

In [257]:

```
### Test and visualize of all preprocessing methods

import matplotlib.image as mpimg
%matplotlib inline

def img_normalization(img):
    img_norm = np.array(img / 255.0 - 0.5 )
    return img_norm



def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])



index = random.randint(0, len(X_train))
image = X_train[index].squeeze()

normalized_image = img_normalization(image)
print('The shape of normalized image is {}'.format(normalized_image.shape))
gray_image = rgb2gray(image)
print('The shape of gray image is {}'.format(gray_image.shape))
normalized_gray_image = img_normalization(gray_image)
print('The shape of normalized gray image is {}'.format(normalized_gray_image.sh
ape))

fig, ax = plt.subplots(1,4, figsize=(10,4))
ax[0].imshow(image)
ax[1].imshow(img_normalization(image))
ax[2].imshow(gray_image, cmap = plt.get_cmap('gray'))
ax[3].imshow(normalized_gray_image, cmap = plt.get_cmap('gray'))
ax[0].set_title('Original Image')
ax[1].set_title('Normalized Image')
ax[2].set_title('Gray Image')
ax[3].set_title('Normalized Gray Image')
print('Image Index ID: {}'.format(index))
print('Class ID: {}'.format(y_train[index]))
print('Sign Name: {}'.format(signname_dict[str(y_train[index])]))
```

```
The shape of normalized image is (32, 32, 3)
The shape of gray image is (32, 32)
The shape of normalized gray image is (32, 32)
Image Index ID: 1004
Class ID: 38
Sign Name: Keep right
```



In [258]:

```
# Image normalization and save it to X_train_norm, X_valid_norm, and X_test_norm

X_train_norm = np.array(X_train / 255.0 - 0.5 )
X_valid_norm = np.array(X_valid / 255.0 - 0.5 )
X_test_norm = np.array(X_test / 255.0 - 0.5 )
```

In [259]:

```
### Convert Color Image to Grayscale

from tqdm import tqdm

"""
X_train_shape = X_train.shape
X_valid_shape = X_valid.shape
X_test_shape = X_test.shape
print(X_train_shape)
print(X_valid_shape)
print(X_test_shape)
"""

X_train_gray_shape = np.array(np.delete(X_train.shape, 3))
X_valid_gray_shape = np.array(np.delete(X_valid.shape, 3))
X_test_gray_shape = np.array(np.delete(X_test.shape, 3))

# print('Converting training data to grayscale...')
images_pbar = tqdm(X_train, unit='images')
X_train_gray_s = np.zeros(X_train_gray_shape)
for i,img in enumerate(images_pbar) :
    X_train_gray_s[i] = rgb2gray(X_train[i])

# print('Converting validation data to grayscale...')
images_pbar = tqdm(X_valid, unit='images')
X_valid_gray_s = np.zeros(X_valid_gray_shape)
for i,img in enumerate(images_pbar) :
    X_valid_gray_s[i] = rgb2gray(X_valid[i])

# print('Converting test data to grayscale......')
images_pbar = tqdm(X_test, unit='images')
X_test_gray_s = np.zeros(X_test_gray_shape)
for i,img in enumerate(images_pbar) :
    X_test_gray_s[i] = rgb2gray(X_test[i])
```

```
100%|██████████████████| 27839/27839 [00:02<00:00, 12477.97images/s]
100%|██████████████████| 6960/6960 [00:00<00:00, 11931.49images/s]
100%|██████████████████| 12630/12630 [00:00<00:00, 12660.92images/s]
```

In [261]:

```
#Resize grayscale images from (None, 32, 32) to (None, 32, 32, 1)
z = tf.placeholder(tf.float32, (None, 32, 32))
size_operation = tf.expand_dims(z, 3)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    X_train_gray = sess.run(size_operation, feed_dict={z: X_train_gray_s})
    X_valid_gray = sess.run(size_operation, feed_dict={z: X_valid_gray_s})
    X_test_gray = sess.run(size_operation, feed_dict={z: X_test_gray_s})
    print('Grayscale training images has successfully been rescaled to size
{}.'.format(X_train.shape))
    print('Grayscale validation images has successfully been rescaled to size
{}.'.format(X_valid.shape))
    print('Grayscale test images has successfully been rescaled to size {}.'.for
mat(X_test.shape))
```

```
Grayscale training images has successfully been rescaled to size (27
839, 32, 32, 3).
Grayscale validation images has successfully been rescaled to size
 (6960, 32, 32, 3).
Grayscale test images has successfully been rescaled to size (12630,
32, 32, 3).
```

In [262]:

```
# Image normalization and save it to X_train_norm, X_valid_norm, and X_test_norm

X_train_ngray = np.array(X_train_gray / 255.0 - 0.5 )
X_valid_ngray = np.array(X_valid_gray / 255.0 - 0.5 )
X_test_ngray = np.array(X_test_gray / 255.0 - 0.5 )
```

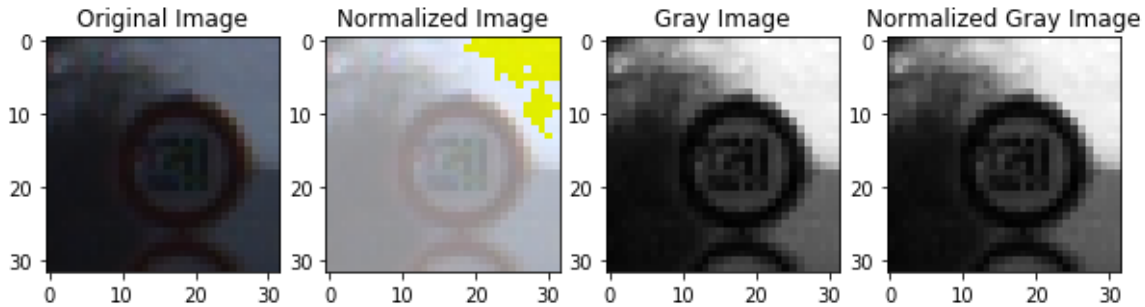
In [263]:

```
"""图像-标签对应测试模块"""
### Test if the image and lables match with each other.
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline


### Show a random image in X_train and print its data

index = random.randint(0, len(X_train))
image = X_train[index].squeeze()
#  _norm   _gray  _ngray
print('Image Index ID: {}'.format(index))
print('Class ID: {}'.format(y_train[index]))
print('Sign Name: {}'.format(signname_dict[str(y_train[index])]))

fig, ax = plt.subplots(1,4, figsize=(10,4))
ax[0].imshow(X_train[index].squeeze())
ax[1].imshow(X_train_norm[index].squeeze())
ax[2].imshow(X_train_gray[index].squeeze(), cmap = plt.get_cmap('gray'))
ax[3].imshow(X_train_ngray[index].squeeze(), cmap = plt.get_cmap('gray'))
ax[0].set_title('Original Image')
ax[1].set_title('Normalized Image')
ax[2].set_title('Gray Image')
ax[3].set_title('Normalized Gray Image')
```

```
Image Index ID: 9603
Class ID: 8
Sign Name: Speed limit (120km/h)
```

Out[263]:

```
<matplotlib.text.Text at 0x1245ae7b8>
```



Model Architecture

In [265]:

```
### Define your architecture here.
### Feel free to use as many code cells as needed.
```

In [266]:

```
def set_prep_parameters(prep_mtd = ['org']):
    #X_train_org, X_valid_org, X_test_org = X_train, X_valid, X_test
    if prep_mtd == 'org':
        X_train_l, X_valid_l, X_test_l = X_train, X_valid, X_test
        input_dim = 3
        print('Parameters set for original images.')
    elif prep_mtd == 'norm':
        X_train_l, X_valid_l, X_test_l = X_train_norm, X_valid_norm, X_test_norm
        input_dim = 3
        print('Parameters set for normalized images.')
    elif prep_mtd == 'gray':
        X_train_l, X_valid_l, X_test_l = X_train_gray, X_valid_gray, X_test_gray
        input_dim = 1
        print('Parameters set for grayscale images.')
    elif prep_mtd == 'ngray':
        X_train_l, X_valid_l, X_test_l = X_train_ngray, X_valid_ngray, X_test_ng
ray
        input_dim = 1
        print('Parameters set for normalized grayscale images.')
    else:
        print('Invalid Preprocessing Method: \'org\' for origional image,
\'norm\' for normalization, \'gray\' for grayscale, \'ngray\' for normalized gra
y scall')
    return X_train_l, X_valid_l, X_test_l, input_dim
```

In [267]:

```
### Parameter Setting Dashboard


# Set training parameters: usu. EPOCHS: 10~50, BATCH_SIZE: 64 or 128
EPOCHS = 23
BATCH_SIZE = 128

# Learning rate during training
rate = 0.0006

# Arguments used for tf.truncated_normal, randomly defines variables for the wei
ghts and biases for each layer
LeNet_mu = 0
LeNet_sigma = 0.1

# Dropout at Fully Connected activation layer: usu. 0.5~0.95
train_keep_prob = 0.6

# Preprocessing Method: 'org' for origional image, 'norm' for normalization, 'gr
ay' for grayscale,
#                       'ngray' for normalized gray scall
prep_mtd = 'norm'
X_train, X_valid, X_test, input_dim = set_prep_parameters(prep_mtd)
# Note: Imput image dimension is set in the above function: 3 for color images,
 1 for gray images

saved_model = './norm_model_2.ckpt'
```

```
Parameters set for normalized images.
```

In [268]:

```
from tensorflow.contrib.layers import flatten

def LeNet(x, keep_prob = [1], input_dim = [3], mu = [0], sigma = [0.1]):
    # Arguments used for tf.truncated_normal, randomly defines variables for the
 weights and biases for each layer
    #mu = 0
    #sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, input_dim, 6), mean =
 mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') +
conv1_b

    # SOLUTION: Activation.
    conv1 = tf.nn.relu(conv1)

    # SOLUTION: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padd
ing='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, st
ddev = sigma))
    conv2_b = tf.Variable(tf.zeros(16))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1],
padding='VALID') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

    # SOLUTION: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padd
ing='VALID')

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0   = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 180.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 180), mean = mu, stddev
= sigma))
    fc1_b = tf.Variable(tf.zeros(180))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

    # SOLUTION: Activation.
    fc1    = tf.nn.relu(fc1)
    fc1  = tf.nn.dropout(fc1, keep_prob)

    # SOLUTION: Layer 4: Fully Connected. Input = 180. Output = 90.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(180, 90), mean = mu, stddev
= sigma))
    fc2_b  = tf.Variable(tf.zeros(90))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

    # SOLUTION: Activation.
    fc2    = tf.nn.relu(fc2)
    fc2  = tf.nn.dropout(fc2, keep_prob)

    # SOLUTION: Layer 5: Fully Connected. Input = 90. Output = 43.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(90, 43), mean = mu, stddev =
```

```
 sigma))
    fc3_b  = tf.Variable(tf.zeros(43))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

In [269]:

```
# Set x, y, one_hot_y
x = tf.placeholder(tf.float32, (None, 32, 32, input_dim))
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 [270]:

```
### 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.
```

In [271]:

```
# Training Pipeline

logits = LeNet(x, keep_prob, input_dim, LeNet_mu, LeNet_sigma)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y,
logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

In [272]:

```
# Model Evaluation
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:offse
t+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_
y, keep_prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [273]:

```
# Train the Model
import time
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)
    accuracy = []

    print("Training...")
    print()

    for i in range(EPOCHS):
        if i == 0:
            start0 = time.time()
        start = time.time()
        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
_prob: train_keep_prob})

        validation_accuracy = evaluate(X_valid, y_valid)
        training_accuracy = evaluate(X_train, y_train)
        accuracy.append(validation_accuracy)
        end = time.time()
        if i%1 == 0 or i == 0:
            print("EPOCH {} ...".format(i+1))
            print("Training Accuracy = {:.3f}".format(training_accuracy))
            print("Validation Accuracy = {:.3f}".format(validation_accuracy))
            print("Speed: {:.2f} s/epoch,".format(end - start), "Time left: {:.1
f} min".format((EPOCHS-i-1)*(end - start0)/(i+1)/60))

    saver.save(sess, saved_model)
    print("Model saved. Total time: {:.1f} min.".format((end - start0)/60))
```

```
Training...

EPOCH 1 ...
Training Accuracy = 0.514
Validation Accuracy = 0.509
Speed: 69.33 s/epoch, Time left: 25.4 min
EPOCH 2 ...
Training Accuracy = 0.782
Validation Accuracy = 0.773
Speed: 62.62 s/epoch, Time left: 23.1 min
EPOCH 3 ...
Training Accuracy = 0.872
Validation Accuracy = 0.865
Speed: 65.80 s/epoch, Time left: 22.0 min
EPOCH 4 ...
Training Accuracy = 0.916
Validation Accuracy = 0.905
Speed: 67.82 s/epoch, Time left: 21.0 min
EPOCH 5 ...
Training Accuracy = 0.937
Validation Accuracy = 0.923
Speed: 67.63 s/epoch, Time left: 20.0 min
EPOCH 6 ...
Training Accuracy = 0.949
Validation Accuracy = 0.936
Speed: 59.39 s/epoch, Time left: 18.5 min
EPOCH 7 ...
Training Accuracy = 0.958
Validation Accuracy = 0.949
Speed: 60.13 s/epoch, Time left: 17.2 min
EPOCH 8 ...
Training Accuracy = 0.967
Validation Accuracy = 0.958
Speed: 55.84 s/epoch, Time left: 15.9 min
EPOCH 9 ...
Training Accuracy = 0.974
Validation Accuracy = 0.961
Speed: 59.85 s/epoch, Time left: 14.7 min
EPOCH 10 ...
Training Accuracy = 0.973
Validation Accuracy = 0.962
Speed: 64.70 s/epoch, Time left: 13.7 min
EPOCH 11 ...
Training Accuracy = 0.978
Validation Accuracy = 0.968
Speed: 60.49 s/epoch, Time left: 12.6 min
EPOCH 12 ...
Training Accuracy = 0.979
Validation Accuracy = 0.971
Speed: 59.47 s/epoch, Time left: 11.5 min
EPOCH 13 ...
Training Accuracy = 0.981
Validation Accuracy = 0.971
Speed: 58.16 s/epoch, Time left: 10.4 min
EPOCH 14 ...
Training Accuracy = 0.986
Validation Accuracy = 0.976
Speed: 58.00 s/epoch, Time left: 9.3 min
EPOCH 15 ...
Training Accuracy = 0.987
Validation Accuracy = 0.975
```

```
Speed: 62.04 s/epoch, Time left: 8.3 min
EPOCH 16 ...
Training Accuracy = 0.989
Validation Accuracy = 0.979
Speed: 60.21 s/epoch, Time left: 7.2 min
EPOCH 17 ...
Training Accuracy = 0.990
Validation Accuracy = 0.980
Speed: 62.06 s/epoch, Time left: 6.2 min
EPOCH 18 ...
Training Accuracy = 0.992
Validation Accuracy = 0.982
Speed: 62.07 s/epoch, Time left: 5.2 min
EPOCH 19 ...
Training Accuracy = 0.993
Validation Accuracy = 0.983
Speed: 64.28 s/epoch, Time left: 4.1 min
EPOCH 20 ...
Training Accuracy = 0.991
Validation Accuracy = 0.981
Speed: 58.00 s/epoch, Time left: 3.1 min
EPOCH 21 ...
Training Accuracy = 0.994
Validation Accuracy = 0.985
Speed: 64.17 s/epoch, Time left: 2.1 min
EPOCH 22 ...
Training Accuracy = 0.994
Validation Accuracy = 0.985
Speed: 59.96 s/epoch, Time left: 1.0 min
EPOCH 23 ...
Training Accuracy = 0.995
Validation Accuracy = 0.984
Speed: 61.99 s/epoch, Time left: 0.0 min
Model saved. Total time: 23.7 min.
```

In [274]:

```
### Recording training results
## Import csv file and read it to a list called data_result
import csv

with open("data_recorder.csv","r") as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    data_result = list(reader)

# data_result = []

def data_record(data_result, prep_mtd, EPOCHS, BATCH_SIZE, train_keep_prob, LeNe
t_mu, LeNet_sigma, learning_rate, accuracy):
    i = len(data_result) + 1
    accuracy = [float(i) for i in accuracy]    # map(float, accuracy) is also go
od
    accuracy = [format(i, '0.4f') for i in accuracy]
    data_result.append([str(i), prep_mtd, EPOCHS, BATCH_SIZE, train_keep_prob, L
eNet_mu, LeNet_sigma, learning_rate, accuracy[-1]])
    return data_result

data_result = data_record(data_result, prep_mtd, EPOCHS, BATCH_SIZE, train_keep_
prob, LeNet_mu, LeNet_sigma, rate, accuracy)
print(data_result)

### Write the result to a CSV file and Save the CSV file

with open("data_recorder.csv", "w") as csv_file:
    writer = csv.writer(csv_file, delimiter=',')
    writer.writerows(data_result)
```

```
[['1', 'norm', '20', '128', '0.6', '0', '0.1', '0.001', '0.9865'],
 ['2', 'norm', '20', '128', '0.5', '0', '0.1', '0.001', '0.9858'],
 ['3', 'gray', '20', '128', '0.5', '0', '0.1', '0.001', '0.9211'],
 ['4', 'gray', '30', '128', '0.5', '0', '0.1', '0.001', '0.9677'],
 ['5', 'ngray', '20', '128', '0.5', '0', '0.1', '0.001', '0.9859'],
 ['6', 'ngray', '30', '128', '0.5', '0', '0.1', '0.0005', '0.9829'],
['7', 'norm', '30', '128', '0.6', '0', '0.1', '0.0006', '0.9879'],
 ['8nm1', 'norm', '25', '128', '0.6', '0', '0.1', '0.0006', '0.985
8'], ['9', 'norm', 23, 128, 0.6, 0, 0.1, 0.0006, '0.9841']]
```

In [275]:

```
# Evaluate the Model
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

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

```
Test Accuracy = 0.928
```

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 [276]:

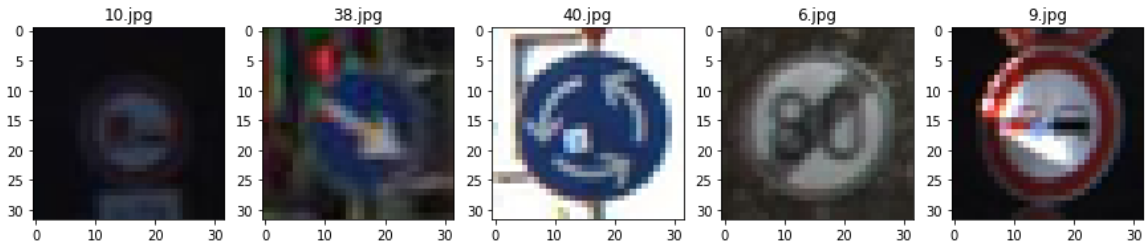
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
```

In [277]:

```
#File conversion
#img = Image.open("Test_data_ppm/005.ppm")
#img.save("Test_data_jpg/005.jpg")
```

In [278]:

```
#from PIL import Image
import PIL
#import matplotlib.image as mpimg
%matplotlib inline
#import matplotlib.pyplot as plt
#import numpy as np



def mylistdir(directory):
    """A specialized version of os.listdir() that ignores files that
    start with a leading period."""
    filelist = os.listdir(directory)
    return [x for x in filelist
            if not (x.startswith('.'))]

img_test_list = []
filename_list = []

for filename in mylistdir('Test_data_jpg'):
    img = PIL.Image.open('Test_data_jpg/' + filename)
    img_test = img.resize((32, 32), 1)
    img_test_list.append(img_test)
    filename_list.append(filename)

fig, ax = plt.subplots(1,len(img_test_list), figsize=(15,15))
for i in range(len(img_test_list)):
    ax[i].imshow(img_test_list[i])
    ax[i].set_title(filename_list[i])

correct_lable_list = [10, 38, 40, 6, 9]
```



Predict the Sign Type for Each Image

In [279]:

```
### 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 us
ed earlier.
### Feel free to use as many code cells as needed.
```

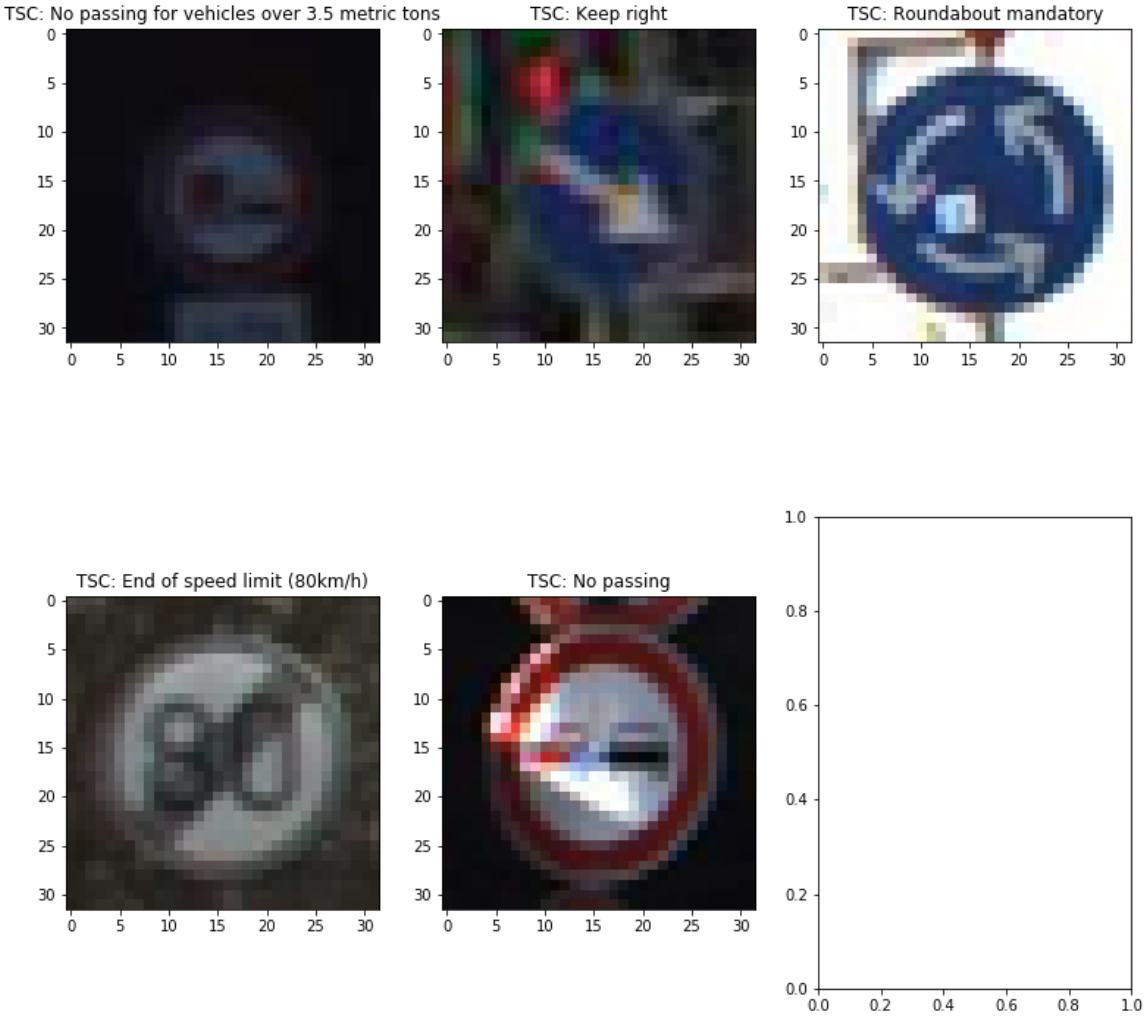
In [280]:

```
#i = 0
#print(correct_lable_list[i])
#print(signname_dict[str(correct_lable_list[i])])
```

In [281]:

```
# Pre-process images
start = time.time()
import math

logits_list = []
i = 0
correct_cnt = 0
%matplotlib inline

def pre_process(img_test_4d_i, prep_mtd):
    if prep_mtd == 'org':
        img_test_4d_o = img_test_4d_i
    elif prep_mtd == 'norm':
        img_test_4d_o = img_normalization(img_test_4d_i)
    elif prep_mtd == 'gray':
        img_test_4d_o = rgb2gray(img_test_4d_i)
    elif prep_mtd == 'ngray':
        img_test_4d_o = img_normalization(rgb2gray(img_test_4d_i))
    else:
        print('Invalid Preprocessing Method: \'org\' for origional image,
\'norm\' for normalization, \'gray\' for grayscale, \'ngray\' for normalized gra
y scall')
    return img_test_4d_o



fig, ax = plt.subplots(math.ceil(len(img_test_list)/3), 3, figsize=(13,13))

for img in img_test_list:
    img_test = np.asarray(img)
    img_test_4d = img_test[np.newaxis, ...]
    img_test_4d_p = pre_process(img_test_4d, prep_mtd)
    with tf.Session() as sess:
        read_model = saved_model  # usu. './model.ckpt'  !!!!!!!!!!!!!!!!!!读取模
型在这里!!!!!!!!!!!!!!!!!!
        saver.restore(sess, read_model)
        logits_new = sess.run(logits, feed_dict = {x: img_test_4d_p, keep_prob:
1.0})
        logits_list.append(logits_new)
        logits_prediction = np.argmax(logits_new)
    ax = plt.subplot(math.ceil(len(img_test_list)/3), 3, i+1)
    plt.imshow(img)
    ax.set_title('TSC: ' + signname_dict[str(logits_prediction)])
    #plt.suptitle('TSC Prediction: ' + signname_dict[str(logits_prediction)])
    if logits_prediction == correct_lable_list[i]:
        correct_cnt += 1
    else:
        print('The image number ' + str(i+1) + ' is incorrect. It should be ' +
signname_dict[str(correct_lable_list[i])] + '.')
    i += 1



end = time.time()
print('Time to run this cell: {:.1f} s'.format(end - start))
print('The accuracy is {0:.0f} %.'.format(100*correct_cnt/len(img_test_list)))
```

```
Time to run this cell: 68.6 s
The accuracy is 100 %.
```



Analyze Performance

In [282]:

```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% acc
urate on these new images.
print('The accuracy is {0:.0f}%.'.format(100*correct_cnt/len(img_test_list)))
```

```
The accuracy is 100%.
```

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,  0.0789
3497,
         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 [283]:

```
### Print out the top five softmax probabilities for the predictions on the Germ
an traffic sign images found on the web.
### Feel free to use as many code cells as needed.
i = 1
logits_soft = tf.placeholder(tf.float32, (n_classes))

with tf.Session() as sess:
    saver.restore(sess, read_model)
    for i_logits in logits_list[0:5]:
        i_logits = np.squeeze(i_logits)
        prob_soft = sess.run(tf.nn.softmax(i_logits), feed_dict = {logits_soft:
i_logits})
        values, indices = sess.run(tf.nn.top_k(tf.constant(prob_soft), k = 5))
        print('Image # %d'%(i), 'top 5 predicted indices are', indices[0:5])
        print('Probabilities are', values[0:5])
        print()
        i += 1
#print(logits_list)
```

```
Image # 1 top 5 predicted indices are [10  9 23  5 42]
Probabilities are [  9.99981642e-01   1.53403034e-05   1.55594182e-0
6   1.00069497e-06
   2.67203291e-07]

Image # 2 top 5 predicted indices are [38 34 20 36 40]
Probabilities are [  1.00000000e+00   6.32613130e-14   1.24744321e-1
5   7.79721125e-19
   2.58768834e-20]

Image # 3 top 5 predicted indices are [40 37 39 33  1]
Probabilities are [  9.82176661e-01   1.78229362e-02   3.60211232e-0
7   4.02539868e-08
   3.73161324e-09]

Image # 4 top 5 predicted indices are [ 6 42 32  5  3]
Probabilities are [  9.99999881e-01   7.47210436e-08   8.57244054e-1
0   5.72543957e-10
   4.39701164e-10]

Image # 5 top 5 predicted indices are [ 9 10 16 42 41]
Probabilities are [  9.99966145e-01   2.27319415e-05   1.10898891e-0
5   2.15331308e-09
   1.72190950e-09]
```

In [284]:

```
a1 = [10,  9,  23,  5, 42]
a2 = [38, 34, 20, 36, 40]
a3 = [40, 37, 39, 33, 4]
a4 = [ 6, 42, 32,  5, 3]
a5 = [ 9, 10, 16, 42, 41]
top5ind = [a1, a2, a3, a4, a5]

for i in range(len(top5ind)):
    ai = top5ind[i]
    print('For image # %d'%(i+1))
    for j in range(len(ai)):
        print('Top # %d predicted indices is'%(j+1),
signname_dict[str(top5ind[i][j])])
```

```
For image # 1
Top # 1 predicted indices is No passing for vehicles over 3.5 metric
tons
Top # 2 predicted indices is No passing
Top # 3 predicted indices is Slippery road
Top # 4 predicted indices is Speed limit (80km/h)
Top # 5 predicted indices is End of no passing by vehicles over 3.5
 metric tons
For image # 2
Top # 1 predicted indices is Keep right
Top # 2 predicted indices is Turn left ahead
Top # 3 predicted indices is Dangerous curve to the right
Top # 4 predicted indices is Go straight or right
Top # 5 predicted indices is Roundabout mandatory
For image # 3
Top # 1 predicted indices is Roundabout mandatory
Top # 2 predicted indices is Go straight or left
Top # 3 predicted indices is Keep left
Top # 4 predicted indices is Turn right ahead
Top # 5 predicted indices is Speed limit (70km/h)
For image # 4
Top # 1 predicted indices is End of speed limit (80km/h)
Top # 2 predicted indices is End of no passing by vehicles over 3.5
 metric tons
Top # 3 predicted indices is End of all speed and passing limits
Top # 4 predicted indices is Speed limit (80km/h)
Top # 5 predicted indices is Speed limit (60km/h)
For image # 5
Top # 1 predicted indices is No passing
Top # 2 predicted indices is No passing for vehicles over 3.5 metric
tons
Top # 3 predicted indices is Vehicles over 3.5 metric tons prohibite
d
Top # 4 predicted indices is End of no passing by vehicles over 3.5
 metric tons
Top # 5 predicted indices is End of no passing
```

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.

Step 4 (Optional): 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)