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 that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points for this project.

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

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

Step 0: Load The Data

In [1]:

```
# Load pickled data
import pickle

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

training_file = "/home/carnd/CarND-Traffic-Sign-Classifier-Project/dataset/train.p"
validation_file="/home/carnd/CarND-Traffic-Sign-Classifier-Project/dataset/valid.p"
testing_file = "/home/carnd/CarND-Traffic-Sign-Classifier-Project/dataset/test.p"

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

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

Step 1: Dataset Summary & Exploration

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

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

- `'sizes'` is a list containing tuples, (width, height) representing the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

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

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

In [2]:

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

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

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

# TODO: Number of testing examples.
n_test = np.shape(X_test)[0]

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

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

print("Number of training examples =", n_train)
print("Number of 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
```

In [3]:

```
import csv

with open('signnames.csv') as csvfile:
    csv_reader = csv.reader(csvfile, delimiter=',')
    signname = []
    for row_id,row in enumerate(csv_reader):
        if row_id == 0:
            pass
        else:
            signname.append(row[1])
```

Include an exploratory visualization of the dataset

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

The Matplotlib examples and gallery pages are a great resource for doing visualizations in Python.

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

In [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
```

```
%matplotlib inline
import os
import cv2

signs_dir = 'signs/'
signs_images = os.listdir(signs_dir)
ground_truth_signs = []
ground_truth_labels = []
for img_name in signs_images:
    img = cv2.imread(signs_dir+img_name)
    ground_truth_signs.append(img[:,:,::-1])
    ground_truth_labels.append(int(img_name.split('.')[0]))
ground_truth_labels = np.array(ground_truth_labels)
```

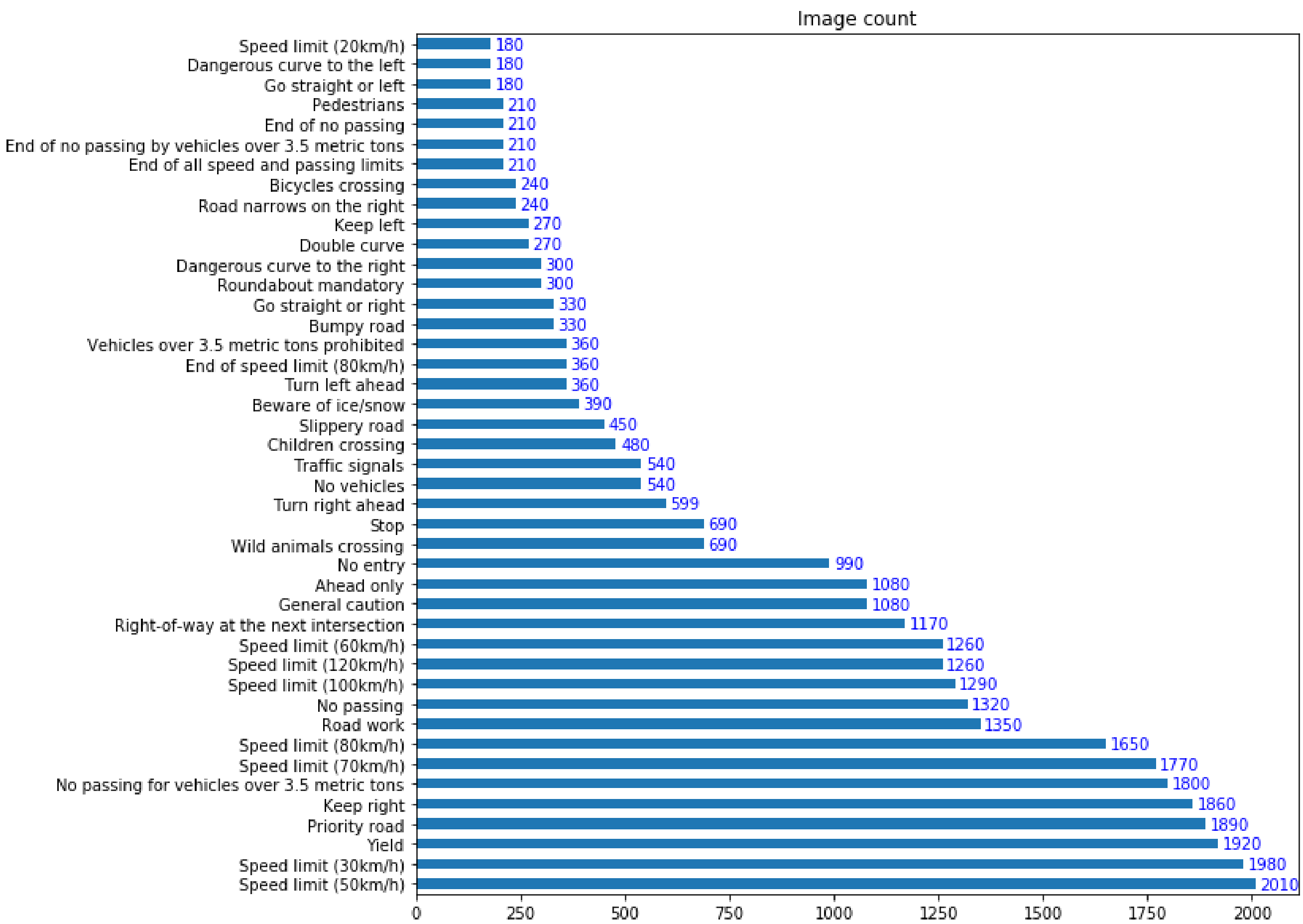
In [5]:

```
import matplotlib.gridspec as gridspec

grid = gridspec.GridSpec(n_classes,5)

data_index = [np.where(np.array(y_train)==classid)[0] for classid in range(n_classes)]

data_grid = plt.figure(num=1,figsize=(15,40))
ax=[]
for classid in range(n_classes):
    ax.append(data_grid.add_subplot(grid[classid,0]))
    ax[-1].text(0, 0.6, signname[classid], ha='left', va='top', wrap=True)
    ax[-1].set_axis_off()

    ax.append(data_grid.add_subplot(grid[classid, 1]))
    gt_image_id = np.where(np.array(ground_truth_labels)==classid)[0][0]
    ax[-1].imshow(ground_truth_signs[gt_image_id])
    ax[-1].set_axis_off()

    index = np.random.choice(data_index[classid], 3,replace=False)

    for sample in range(3):
        image = X_train[index[sample]].squeeze()
        ax.append(data_grid.add_subplot(grid[classid, sample+2]))
        ax[-1].imshow(image)
        ax[-1].set_axis_off()
```


Right-of-way at the next intersection
Priority road
Yield
Stop
STOP
No vehicles
Vehicles over 3.5 metric tons prohibited
No entry
General caution
Dangerous curve to the left
Dangerous curve to the right
Double curve
Bumpy road
Slippery road
Road narrows on the right
Road work
Traffic signals
Pedestrians
Children crossing
Bicycles crossing
Beware of ice/snow
Wild animals crossing
End of all speed and passing limits
Turn right ahead
Turn left ahead
Ahead only

In [6]:

```python
import pandas as pd

def plot_histogram(y_train):
    classes = pd.DataFrame()
    classes['label'] = y_train
    ax = classes['label'].value_counts().plot(kind='barh', figsize = (10,10), title='Image count')
    ax.set_yticklabels(list(map(lambda x: signname[x], classes['label'].value_counts().index.tolist()))
)

    for i,v in enumerate(classes['label'].value_counts()):
        ax.text(v + 10, i - 0.25, str(v), color='blue')

plot_histogram(y_train)
```



Step 2: Design and Test a Model Architecture

Step 2: Design and Test a Model Architecture

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

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

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

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

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

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

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.

Question 1

Describe the techniques used to preprocess the data.

Answer:

The above barchart shows "Number of examples per label" in original training dataset. Some labels are lesser in number,some are more in number,so to normalize the dataset we generate some data for each label

1. Merge provide train set and test set together, yeild 39209+12630= 51839 real samples
2. Rotate the sample image in range +/-20 degrees, generate additional samples for underrepresented classes This step will bring those rare samples (such as 20km/h, Pedestrian, etc) up to 2000+ level.

Then split the total dataset 80:20 for trainset and testset.With in the trainset, split 75:25 again for training set and validation set. Therefore, I can train the model on 61683 samples, validate on 20562 samples, test on 20562 samples.

In [7]:

```
print('Merge training set and testing set...')

X_all = np.concatenate((X_train, X_test), axis = 0)
y_all = np.concatenate((y_train, y_test), axis = 0)
X1_all = np.concatenate((X_all, X_valid), axis = 0)
y1_all = np.concatenate((y_all, y_valid), axis = 0)
print('Merge completed. Number of total samples', len(y1_all))
```

```
Merge training set and testing set...
Merge completed. Number of total samples 51839
```

Question 2

Describe how you set up the training, validation and testing data for your model. If you generated additional data, why?

Answer:

I almost doubled the total samples from original samples to total 102807 samples. You can see some rotated images are showing. It will take care some cases like camera shaking or place at not perfect angle. For future work, maybe can add tilt, warp and shift to the image. The dataset can be easily grow 5-10 times bigger.

For this project, I am using 80:20 split on total dataset to get testing set samples. Then split the training set 75:25 again to get training set 61683 samples and validation set 20562 samples.

Even in validation set, the lowest image count per class (such as stop sign) is over 200, greater than "30 rule", I am ok to proceed with these setting.

In [8]:

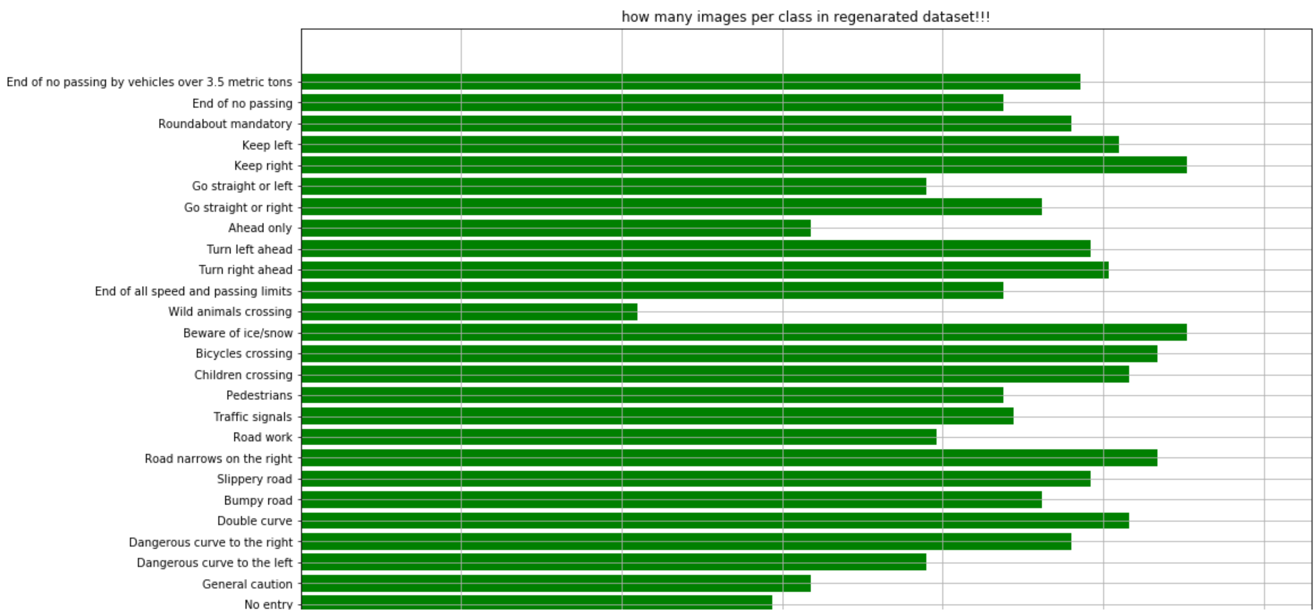
```
import scipy.ndimage
class_count = np.bincount(y1_all)
# Generate additional data for underrepresented classes
print('Generating additional data...')
angles = [-5, 5, -10, 10, -15, 15, -20, 20]

for i in range(len(class_count)):
    input_ratio = min(int(np.max(class_count) / class_count[i]) - 1, len(angles) - 1)

    if input_ratio <= 1:
        continue

    new_features = []
    new_labels = []
    mask = np.where(y_all == i)

    for j in range(input_ratio):
        for feature in X_all[mask]:
            new_features.append(scipy.ndimage.rotate(feature, angles[j], reshape=False))
            new_labels.append(i)

    X1_all = np.append(X1_all, new_features, axis=0)
    y1_all = np.append(y1_all, new_labels, axis=0)

print('Regenarating data completed. Number of total samples', len(y1_all))
```

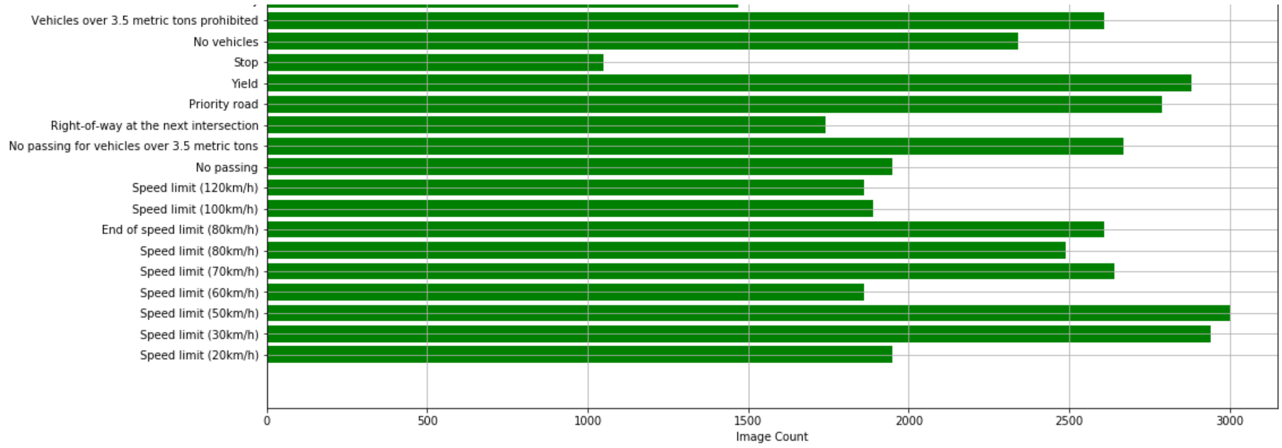
```
Generating additional data...
Regenarating data completed. Number of total samples 97497
```

In [21]:

```
print('Counting samples per class...')
from pylab import *
class_count = np.bincount(y1_all)
pos = arange(43)+.5    # the bar centers on the y axis

figure(111,figsize = (16,16))
barh(pos,class_count, align='center',color='green')
yticks(pos, signname,)
xlabel('Image Count')
title('how many images per class in regenarated dataset!!!')
grid(True)

show()
```

```
Counting samples per class...
```





In [10]:

```
from sklearn.utils import shuffle

X1_all, y1_all = shuffle(X1_all, y1_all)
```

In [11]:

```
print('splitting the total data into training/validation/testing sets...')
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X1_all, y1_all, test_size=0.2, stratify = y1_all )
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25, stratify = y_train
)
print('Number of training samples and size', X_train.shape)
print('Number of validation samples and size', X_val.shape)
print('Number of testing samples and size', X_test.shape)
```

```
splitting the total data into training/validation/testing sets...
Number of training samples and size (58497, 32, 32, 3)
Number of validation samples and size (19500, 32, 32, 3)
Number of testing samples and size (19500, 32, 32, 3)
```

In [12]:

```
import tensorflow as tf
from spatial_transformer import transformer  # ref:https://github.com/tensorflow/models/blob/master/tra
nsformer/spatial_transformer.py
from tf_utils import weight_variable, bias_variable, dense_to_one_hot
from tensorflow.contrib.layers import flatten
```

In [13]:

```
from sklearn.utils import shuffle
X_train, y_train = shuffle(X_train, y_train)

### Calculate Mean image to be used for normlization of Network input

mean_image = np.mean(X_train,axis=0)
```

Model Architecture

Localization Network

In [14]:

```
x = tf.placeholder(tf.float32, (None, 32, 32, 3))
mean_image_placeholder = tf.placeholder(tf.float32, (32, 32, 3))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
keep_prob = tf.placeholder(tf.float32)
keep_prob_fc = tf.placeholder(tf.float32)
keep_prob_conv = tf.placeholder(tf.float32)
```

In [15]:

```
# Identity transformation
```

```
initial = np.array([[1., 0, 0], [0, 1., 0]])
initial = initial.astype('float32')
initial = initial.flatten()

# Create variables for fully connected layer for the localisation network
W_fc_loc1 = weight_variable([600, 100])
b_fc_loc1 = bias_variable([100])

W_fc_loc2 = weight_variable([100, 6])
b_fc_loc2 = tf.Variable(initial_value=initial, name='b_fc_loc2')
```

In [16]:

```
def localization_net(x):
    mu = 0
    sigma = 0.1
    loc_conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 6), mean = mu, stddev = sigma))
    loc_conv1_b = tf.Variable(tf.zeros(6))
    # Layer 1: Convolutional. Output = 28x28x6.
    loc_conv1   = tf.nn.conv2d(x, loc_conv1_W, strides=[1, 1, 1, 1], padding='VALID') + loc_conv1_b

    # layer1: Activation.
    loc_conv1 = tf.nn.relu(loc_conv1)

    # layer1: Pooling. Input = 28x28x6. Output = 14x14x6.
    loc_conv1 = tf.nn.max_pool(loc_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # Layer 2: Convolutional. Output = 10x10x16.
    loc_conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 24), mean = mu, stddev = sigma))
    loc_conv2_b = tf.Variable(tf.zeros(24))
    loc_conv2   = tf.nn.conv2d(loc_conv1, loc_conv2_W, strides=[1, 1, 1, 1], padding='VALID') + loc_con
v2_b

    # Layer 2: Activation.
    loc_conv2 = tf.nn.relu(loc_conv2)

    # Layer 2: Pooling. Input = 10x10x24. Output = 5x5x24.
    loc_conv2 = tf.nn.max_pool(loc_conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # Layer 3: Flatten. Input = 5x5x24. Output = 400.
    fc0   = flatten(loc_conv2)
    # Layer 4: Fully connected layers
    h_fc_loc1 = tf.nn.tanh(tf.matmul(fc0, W_fc_loc1) + b_fc_loc1)

    # Layer 5: Dropout for regularization
    h_fc_loc1_drop = tf.nn.dropout(h_fc_loc1, keep_prob)

    # Layer 6: Fully connected layers
    h_fc_loc2 = tf.nn.tanh(tf.matmul(h_fc_loc1_drop, W_fc_loc2) + b_fc_loc2)
    return h_fc_loc2
```

In [17]:

```
def inception_block(input,features_num):
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional 1x1
    conv1x1_w = tf.Variable(tf.truncated_normal(shape=(1, 1, features_num, 24), mean = mu, stddev = sig
ma))
    conv1x1_b = tf.Variable(tf.zeros(24))
    conv1x1   = tf.nn.conv2d(input, conv1x1_w, strides=[1, 3, 3, 1], padding='SAME') + conv1x1_b
    conv1x1 = tf.nn.relu(conv1x1)

    # Layer 2: Convolutional 3x3
    conv3x3_w = tf.Variable(tf.truncated_normal(shape=(3, 3, 24, 16), mean = mu, stddev = sigma))
    conv3x3_b = tf.Variable(tf.zeros(16))
    conv3x3   = tf.nn.conv2d(conv1x1, conv3x3_w, strides=[1, 1, 1, 1], padding='SAME') + conv3x3_b
    conv3x3 = tf.nn.relu(conv3x3)

    # Layer 3: Convolutional 5x5
    conv5x5_w = tf.Variable(tf.truncated_normal(shape=(5, 5, 24, 8), mean = mu, stddev = sigma))
    conv5x5_b = tf.Variable(tf.zeros(8))
    conv5x5   = tf.nn.conv2d(conv1x1, conv5x5_w, strides=[1, 1, 1, 1], padding='SAME') + conv5x5_b
    conv5x5 = tf.nn.relu(conv5x5)

    # Layer 4: max pooling 3x3
```

```
    max3x3   = tf.nn.max_pool(conv3x3, ksize=[1, 3, 3, 1], strides=[1, 1, 1, 1], padding='SAME')
    # Concatenates feature maps
    output = tf.concat(3,[conv1x1, conv3x3,conv5x5,max3x3])
    return output
```

In [18]:

```
def unified_inception_model(x):
    mu = 0
    sigma = 0.1

    # normlization layer
    # batch_input = x - mean_image_placeholder
    """
    Input Normalization was tried but tends to give less accuracy than using input data as it
    """
    batch_input = x

    ## Create a spatial transformer module
    affain_transformation = localization_net(batch_input)
    x_trans = transformer(batch_input, affain_transformation, (32, 32))

    # inception Block #1

    inception_1 = inception_block(x_trans,3)

    # Dropout layer

    inceptoon_dropout = tf.nn.dropout(inception_1, keep_prob_conv)

    # inception Block #2

    inception_2 = inception_block(inception_1,64)

    # Flatten layer

    flatten_features    = flatten(inception_2)

    # Fully connected layer 1.. Input = 1000. Output = 1000.

    fc1_W = tf.Variable(tf.truncated_normal(shape=(1024, 512), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(512))
    fc1   = tf.matmul(flatten_features, fc1_W) + fc1_b
    fc1    = tf.nn.relu(fc1)

    # Fully connected layer 2. Input = 512. Output = 1000.

    fc2_W  = tf.Variable(tf.truncated_normal(shape=(512, 100), mean = mu, stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(100))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b
    fc2    = tf.nn.relu(fc2)

    # Dropout for regularization

    fc2_droped = tf.nn.dropout(fc2, keep_prob_fc)

    # Fully connected layer3. Input = 100. Output = n_classes.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(100, n_classes), mean = mu, stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros(n_classes))
    logits = tf.matmul(fc2_droped, fc3_W) + fc3_b

    return inception_1, inception_2, logits , x_trans
```

Train, Validate and Test the Model

In [19]:

```
EPOCHS = 50
BATCH_SIZE = 128
rate = 0.0001
# Training pipeline

inception_1, inception_2, logits , x_trans = unified_inception_model(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits)
network_output = tf.nn.softmax(logits=logits)
```

```
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

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 [20]:

```
### 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.

# Validation pipeline
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+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y,mean_image_placeholde
r:mean_image, keep_prob: 1.0, keep_prob_fc: 1.0, keep_prob_conv: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [22]:

```
#training phase
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)
    training_accuracy = []
    validation_accuracy = []
    print("Training...")
    print()
    for i in range(EPOCHS):
        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, mean_image_placeholder:mean
_image, keep_prob: 0.4, keep_prob_fc: 0.5, keep_prob_conv: 0.5})

        training_accuracy.append(evaluate(X_train, y_train))
        validation_accuracy.append(evaluate(X_val, y_val))

        print("EPOCH {} ...".format(i+1))
        print("Training Accuracy = {:.3f}".format(training_accuracy[-1]))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy[-1]))
        print()

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

```
Training...

EPOCH 1 ...
Training Accuracy = 0.163
Validation Accuracy = 0.161

EPOCH 2 ...
Training Accuracy = 0.362
Validation Accuracy = 0.363

EPOCH 3 ...
Training Accuracy = 0.517
Validation Accuracy = 0.517

EPOCH 4 ...
Training Accuracy = 0.643
Validation Accuracy = 0.639
```

```
EPOCH 5 ...
Training Accuracy = 0.710
Validation Accuracy = 0.709

EPOCH 6 ...
Training Accuracy = 0.804
Validation Accuracy = 0.802

EPOCH 7 ...
Training Accuracy = 0.848
Validation Accuracy = 0.843

EPOCH 8 ...
Training Accuracy = 0.883
Validation Accuracy = 0.880

EPOCH 9 ...
Training Accuracy = 0.909
Validation Accuracy = 0.907

EPOCH 10 ...
Training Accuracy = 0.904
Validation Accuracy = 0.901

EPOCH 11 ...
Training Accuracy = 0.940
Validation Accuracy = 0.935

EPOCH 12 ...
Training Accuracy = 0.951
Validation Accuracy = 0.945

EPOCH 13 ...
Training Accuracy = 0.961
Validation Accuracy = 0.956

EPOCH 14 ...
Training Accuracy = 0.967
Validation Accuracy = 0.962

EPOCH 15 ...
Training Accuracy = 0.965
Validation Accuracy = 0.958

EPOCH 16 ...
Training Accuracy = 0.969
Validation Accuracy = 0.965

EPOCH 17 ...
Training Accuracy = 0.977
Validation Accuracy = 0.972

EPOCH 18 ...
Training Accuracy = 0.979
Validation Accuracy = 0.974

EPOCH 19 ...
Training Accuracy = 0.982
Validation Accuracy = 0.978

EPOCH 20 ...
Training Accuracy = 0.984
Validation Accuracy = 0.980

EPOCH 21 ...
Training Accuracy = 0.983
Validation Accuracy = 0.979

EPOCH 22 ...
Training Accuracy = 0.987
Validation Accuracy = 0.983

EPOCH 23 ...
Training Accuracy = 0.987
Validation Accuracy = 0.983
```

```
EPOCH 24 ...
Training Accuracy = 0.986
Validation Accuracy = 0.983

EPOCH 25 ...
Training Accuracy = 0.985
Validation Accuracy = 0.981

EPOCH 26 ...
Training Accuracy = 0.990
Validation Accuracy = 0.986

EPOCH 27 ...
Training Accuracy = 0.990
Validation Accuracy = 0.987

EPOCH 28 ...
Training Accuracy = 0.990
Validation Accuracy = 0.986

EPOCH 29 ...
Training Accuracy = 0.992
Validation Accuracy = 0.988

EPOCH 30 ...
Training Accuracy = 0.993
Validation Accuracy = 0.990

EPOCH 31 ...
Training Accuracy = 0.989
Validation Accuracy = 0.986

EPOCH 32 ...
Training Accuracy = 0.992
Validation Accuracy = 0.989

EPOCH 33 ...
Training Accuracy = 0.993
Validation Accuracy = 0.990

EPOCH 34 ...
Training Accuracy = 0.994
Validation Accuracy = 0.991

EPOCH 35 ...
Training Accuracy = 0.994
Validation Accuracy = 0.991

EPOCH 36 ...
Training Accuracy = 0.995
Validation Accuracy = 0.992

EPOCH 37 ...
Training Accuracy = 0.995
Validation Accuracy = 0.993

EPOCH 38 ...
Training Accuracy = 0.994
Validation Accuracy = 0.991

EPOCH 39 ...
Training Accuracy = 0.994
Validation Accuracy = 0.991

EPOCH 40 ...
Training Accuracy = 0.995
Validation Accuracy = 0.992

EPOCH 41 ...
Training Accuracy = 0.996
Validation Accuracy = 0.994

EPOCH 42 ...
Training Accuracy = 0.993
Validation Accuracy = 0.991

EPOCH 43 ...
```

```
Training Accuracy = 0.996
Validation Accuracy = 0.993

EPOCH 44 ...
Training Accuracy = 0.996
Validation Accuracy = 0.994

EPOCH 45 ...
Training Accuracy = 0.996
Validation Accuracy = 0.993

EPOCH 46 ...
Training Accuracy = 0.997
Validation Accuracy = 0.994

EPOCH 47 ...
Training Accuracy = 0.997
Validation Accuracy = 0.995

EPOCH 48 ...
Training Accuracy = 0.996
Validation Accuracy = 0.993

EPOCH 49 ...
Training Accuracy = 0.997
Validation Accuracy = 0.994

EPOCH 50 ...
Training Accuracy = 0.997
Validation Accuracy = 0.995

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 [28]:

```python
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import os
test_images_dir = "test_images/"
test_images=os.listdir(test_images_dir)
images = []
labels = []
f, axarr = plt.subplots(4, 2)
for id,img_name in enumerate(test_images):
    img = cv2.imread(test_images_dir+img_name)
    images.append(img[:,:,::-1])
    labels.append(int(img_name.split('.')[0]))
    axarr[id%4, id // 4].imshow(images[-1])
    axarr[id%4, id // 4].set_title(signname[labels[-1]])
    axarr[id%4, id // 4].set_axis_off()
f.set_size_inches(5,15)
f.subplots_adjust(wspace=1, hspace=0.1)
```

Keep right

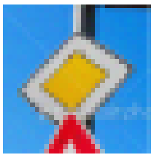
Right-of-way at the next intersection

Speed limit (60km/h)

Road work

Speed limit (30km/h)

Priority road



Predict the Sign Type for Each Image

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

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

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

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the 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. `tf.nn.top_k` is used to choose the three classes with the highest probability:

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

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

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

```
        [0, 1, 4],
        [0, 5, 1],
        [1, 3, 5],
        [1, 4, 3]], dtype=int32))
```

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

In [29]:

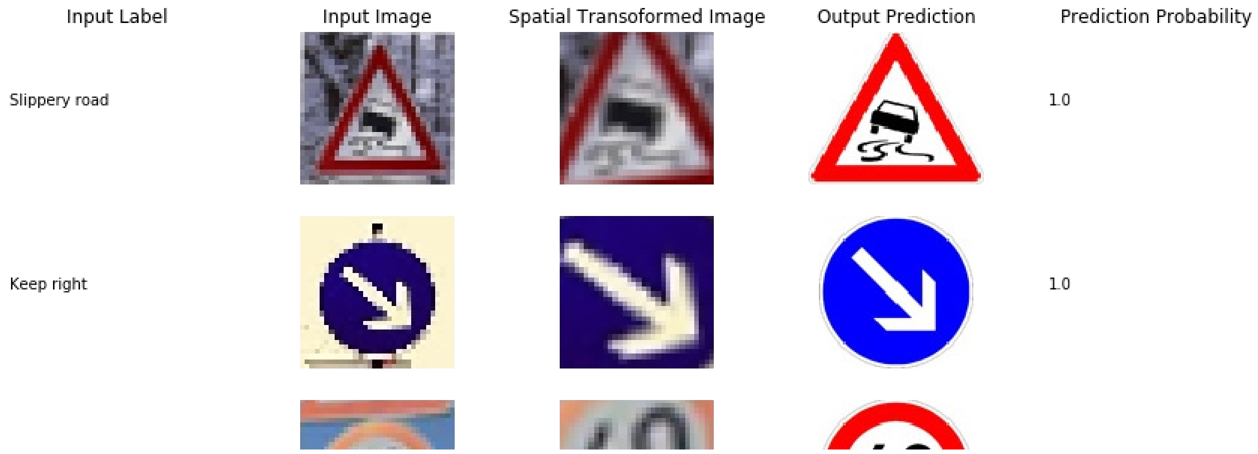
```python
### Run the predictions here and use the model to output the prediction for each image.
### Make sure to pre-process the images with the same pre-processing pipeline used earlier.
### Feel free to use as many code cells as needed.
def forward_inference(X_data, y_data):
    sess = tf.get_default_session()
    accuracy, transformed_image = sess.run([network_output,x_trans], feed_dict={x: X_data, y: y_data, m
ean_image_placeholder:mean_image, keep_prob: 1.0, keep_prob_fc: 1.0, keep_prob_conv: 1.0})
    return accuracy, transformed_image


with tf.Session() as sess:
    # Restore variables from disk.
    output_probabilites = []
    transformed_images = []
    predicted_labels = []
    saver.restore(sess, "./lenet")
    print("Model restored.")
    for i in range(len(images)):
        images_TF = tf.expand_dims(images[i], 0)
        img_prob,img_transformed = forward_inference(images_TF.eval(),tf.expand_dims(np.array(labels[i]
), 0).eval())
        output_probabilites.append(img_prob)
        transformed_images.append(img_transformed)
        predicted_labels.append((np.argmax(output_probabilites[i][0]),np.max(output_probabilites[i][0])
))
        print("model predicted \"{}\" with accuracy {:.5f} but True label is \" {}\" with accuracy {:.5
f} ".format(signname[np.argmax(output_probabilites[i][0])],
                                                                                np.max(output_probabilites[i][0]),signname
[labels[i]],output_probabilites[i][0][labels[i]]))

    TopKV5 = sess.run(tf.nn.top_k(tf.constant(np.array(output_probabilites)), k=5))
    print(TopKV5)
```

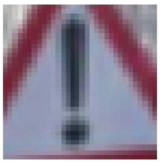
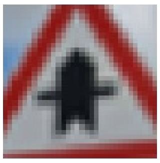
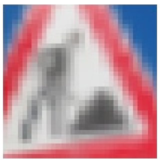
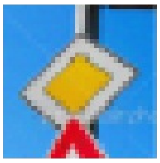
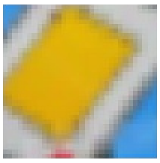
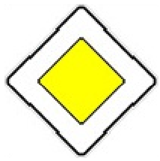
```
Model restored.
model predicted "Slippery road" with accuracy 1.00000 but True label is " Slippery road" with accuracy
1.00000
model predicted "Keep right" with accuracy 1.00000 but True label is " Keep right" with accuracy 1.0000
0
model predicted "Speed limit (60km/h)" with accuracy 0.99998 but True label is " Speed limit (60km/h)"
with accuracy 0.99998
model predicted "Speed limit (30km/h)" with accuracy 0.88322 but True label is " Speed limit (30km/h)"
with accuracy 0.88322
model predicted "General caution" with accuracy 1.00000 but True label is " General caution" with accur
acy 1.00000
model predicted "Right-of-way at the next intersection" with accuracy 1.00000 but True label is " Right
-of-way at the next intersection" with accuracy 1.00000
model predicted "Road work" with accuracy 1.00000 but True label is " Road work" with accuracy 1.00000
model predicted "Priority road" with accuracy 1.00000 but True label is " Priority road" with accuracy
1.00000
TopKV2(values=array([[[  1.00000000e+00,   1.41531744e-17,   1.34468021e-18,
           7.75606470e-19,   1.57665407e-19]],

       [[  1.00000000e+00,   1.33198870e-14,   7.99712062e-17,
           3.52446852e-18,   6.88299946e-22]],

       [[  9.99980927e-01,   9.97891129e-06,   9.12881660e-06,
           7.63093200e-10,   1.79751200e-10]],

       [[  8.83219540e-01,   1.03569441e-01,   1.29050631e-02,
           1.80185001e-04,   9.69376633e-05]],

       [[  1.00000000e+00,   6.93111974e-12,   3.01913849e-14,
           1.25542727e-19,   6.32123589e-21]],

       [[  1.00000000e+00,   5.65436196e-12,   2.15475024e-20,
           8.26572085e-24,   2.03905280e-24]],
```

```
       [[  1.00000000e+00,   3.22312420e-11,   1.78620192e-11,
           2.14498914e-14,   1.12103633e-16]],

       [[  1.00000000e+00,   7.46231027e-37,   0.00000000e+00,
           0.00000000e+00,   0.00000000e+00]]], dtype=float32), indices=array([[[23, 30, 19, 10, 11]],

       [[38, 40, 34, 20, 35]],

       [[ 3,  5,  2, 16,  9]],

       [[ 1,  2,  5,  4,  3]],

       [[18, 26, 27, 24, 20]],

       [[11, 30, 27, 40, 18]],

       [[25, 22, 29, 30, 24]],

       [[12, 13,  0,  1,  2]]], dtype=int32))
```

In [30]:

```python
test_gs = gridspec.GridSpec(len(images)+1, 5)

# plot samples for each traffic signs
data_set_fig = plt.figure(figsize=(15, 20))
# st = data_set_fig.suptitle("Training  DataSet Samples", fontsize="x-large")
ax = []
for img_id in range(len(images)):
    # plot Input Image and label
    ax.append(data_set_fig.add_subplot(test_gs[img_id, 0]))
    ax[-1].text(0, 0.6, signname[labels[img_id]], ha='left', va='top', wrap=True)
    ax[-1].set_axis_off()
    if img_id == 0:
        plt.title('Input Label')
    ax.append(data_set_fig.add_subplot(test_gs[img_id, 1]))
    ax[-1].imshow(images[img_id])
    ax[-1].set_axis_off()
    if img_id == 0:
        plt.title('Input Image')
    # plot Transformation of output
    ax.append(data_set_fig.add_subplot(test_gs[img_id, 2]))
    ax[-1].imshow(np.array(transformed_images[img_id][0],dtype=np.uint8))
    ax[-1].set_axis_off()
    if img_id == 0:
        plt.title('Spatial Transoformed Image')
    #plot predected sign image and accuarcy
    ax.append(data_set_fig.add_subplot(test_gs[img_id, 3]))
    gt_image_id = np.where(np.array(ground_truth_labels)==predicted_labels[img_id][0])[0][0]
    ax[-1].imshow(ground_truth_signs[gt_image_id])
    ax[-1].set_axis_off()
    if img_id == 0:
        plt.title('Output Prediction')
    ax.append(data_set_fig.add_subplot(test_gs[img_id, 4]))
    ax[-1].text(0, 0.6, str(predicted_labels[img_id][1]), ha='left', va='top', wrap=True)
    ax[-1].set_axis_off()
    if img_id == 0:
        plt.title('Prediction Probability')
```



Speed limit (60km/h)				0.999981
Speed limit (30km/h)				0.88322
General caution				1.0
Right-of-way at the next intersection				1.0
Road work				1.0
Priority road				1.0

Analyze Performance

In [36]:

```
### 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 imag
es.
overall_accuracy = 0
for sample_id, sample_result in enumerate(output_probabilites):
    if np.argmax(sample_result[0]) == labels[sample_id]:
        overall_accuracy += 1
overall_accuracy = overall_accuracy / float(len(output_probabilites))
print("Model Accuracy over small test set = {:.2f} %".format(overall_accuracy*100))
```

```
Model Accuracy over small test set = 100.00 %
```

In [35]:

```
with tf.Session() as sess:
    # Restore variables from disk.
    saver.restore(sess, "./lenet")
    print(evaluate(X_test,y_test))
```

```
0.994666666667
```

Project Writeup

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

> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the

> menu above and navigating to \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] 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] 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.

Combined Image

Your output should look something like this (above)

In [ ]:

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

# image_input: the test image being fed into the network to produce the feature maps
# tf_activation: should be a tf variable name used during your training procedure that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more detail, by default 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 on the same block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network expects
    # with size, normalization, ect if needed
    # image_input =
    # 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 may be having trouble accessing the variable from inside a function
    activation = tf_activation.eval(session=sess,feed_dict={x : image_input})
    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 feature map number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap="gray")
```