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]:

```
# Load pickled data
import cv2
import pickle
import matplotlib.pyplot as plt
import pandas as pd
import sklearn
import tensorflow as tf
import numpy as np
import os
from matplotlib import __version__ as plt__version
import sys
from math import ceil, pow
from tqdm import tqdm
from sklearn.metrics import confusion_matrix


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

training_file = 'dataset/train.p'
validation_file= 'dataset/valid.p'
testing_file = '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)

train_f, train_l = train['features'], train['labels']
valid_f, valid_l = valid['features'], valid['labels']
test_f, test_l = test['features'], test['labels']

signnames = pd.read_csv("signnames.csv").values[:, 1]
sign_classes, class_indices, class_counts = np.unique(train_l, return_index = True, ret
urn_counts = True)

""" Check whether the number of features and labels is the same. """
assert(len(train_f) == len(train_l))
assert(len(valid_f) == len(valid_l))
```

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 [2]:

```
# TODO: Number of training examples
train_size = len(train_f)
valid_size = len(valid_f)
test_size = len(test_f)

image_shape = train_f[0].shape
classes_size = len(np.unique(train_l))

print("Number of training examples = ", train_size)
print("Number of validation examples = ", valid_size)
print("Number of testing examples = ", test_size)
print("Number of classes = ", classes_size)
print("Image data shape = ", image_shape)
```

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

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 [3]:

```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt

# Plot image class distribution

values, counts = np.unique(train_l, return_counts=True)
plt.figure(figsize=(20, 5))
plt.bar(values, counts)
plt.xlabel('Classes')
plt.ylabel('Images')
plt.title("Training dataset ditribution")
```

Out[3]:

```
Text(0.5,1,'Training dataset ditribution')
```

In [4]:

```
import random

col_width = max(len(name) for name in signnames)

for c, i, count in zip(class_indices, sign_classes, class_counts):
    print("[Class %i, %s samples]: %s" % (i, str(count), signnames[i]))
    fig = plt.figure(figsize = (8, 1))
    fig.subplots_adjust(left = 0, right = 1, bottom = 0, top = 1, hspace = 0.05, wspace
 = 0.05)
    random_indices = random.sample(range(c, c + count), 8)
    for i in range(8):
        axis = fig.add_subplot(1, 8, i + 1, xticks=[], yticks=[])
        axis.imshow(train_f[random_indices[i]])
    plt.show()
```

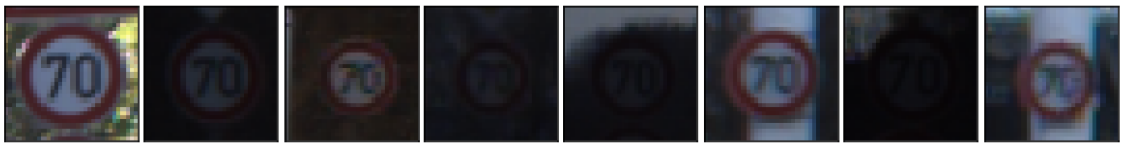
[Class 0, 180 samples]: Speed limit (20km/h)

[Class 1, 1980 samples]: Speed limit (30km/h)

[Class 2, 2010 samples]: Speed limit (50km/h)

[Class 3, 1260 samples]: Speed limit (60km/h)

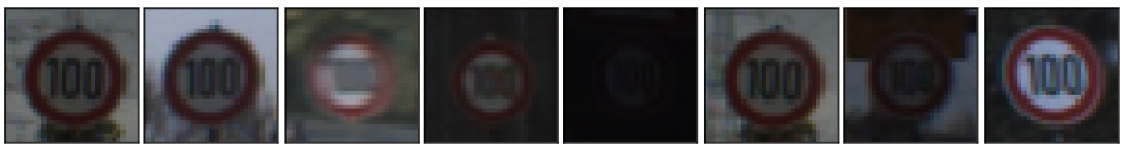
[Class 4, 1770 samples]: Speed limit (70km/h)



[Class 5, 1650 samples]: Speed limit (80km/h)

[Class 6, 360 samples]: End of speed limit (80km/h)

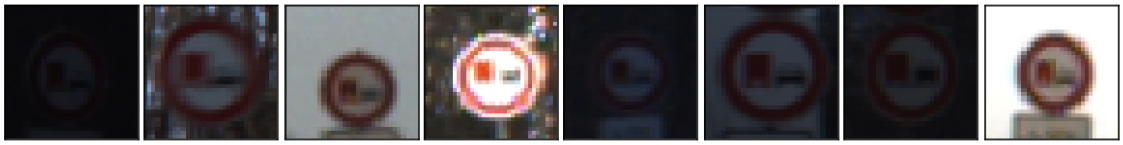
[Class 7, 1290 samples]: Speed limit (100km/h)



[Class 8, 1260 samples]: Speed limit (120km/h)

[Class 9, 1320 samples]: No passing

[Class 10, 1800 samples]: No passing for vehicles over 3.5 metric tons



[Class 11, 1170 samples]: Right-of-way at the next intersection

[Class 12, 1890 samples]: Priority road

[Class 13, 1920 samples]: Yield

[Class 14, 690 samples]: Stop

[Class 15, 540 samples]: No vehicles

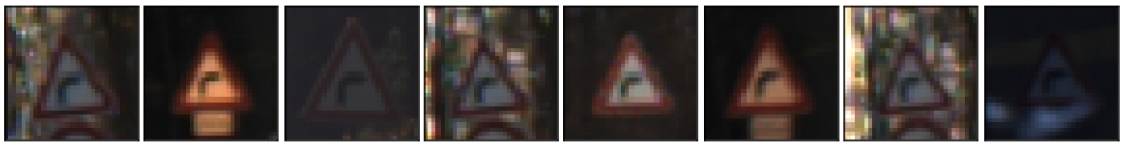
[Class 16, 360 samples]: Vehicles over 3.5 metric tons prohibited

[Class 17, 990 samples]: No entry

[Class 18, 1080 samples]: General caution

[Class 19, 180 samples]: Dangerous curve to the left

[Class 20, 300 samples]: Dangerous curve to the right



[Class 21, 270 samples]: Double curve

[Class 22, 330 samples]: Bumpy road

[Class 23, 450 samples]: Slippery road

[Class 24, 240 samples]: Road narrows on the right

[Class 25, 1350 samples]: Road work

[Class 26, 540 samples]: Traffic signals

[Class 27, 210 samples]: Pedestrians

[Class 28, 480 samples]: Children crossing

[Class 29, 240 samples]: Bicycles crossing

[Class 30, 390 samples]: Beware of ice/snow

[Class 31, 690 samples]: Wild animals crossing

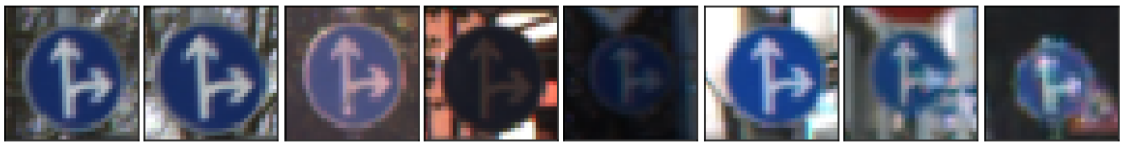
[Class 32, 210 samples]: End of all speed and passing limits

[Class 33, 599 samples]: Turn right ahead

[Class 34, 360 samples]: Turn left ahead

[Class 35, 1080 samples]: Ahead only

[Class 36, 330 samples]: Go straight or right



[Class 37, 180 samples]: Go straight or left

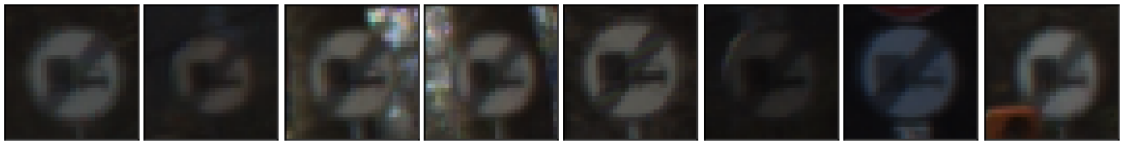
[Class 38, 1860 samples]: Keep right

[Class 39, 270 samples]: Keep left

[Class 40, 300 samples]: Roundabout mandatory

```
[Class 41, 210 samples]: End of no passing
```

```
[Class 42, 210 samples]: End of no passing by vehicles over 3.5 metric ton
s
```



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.

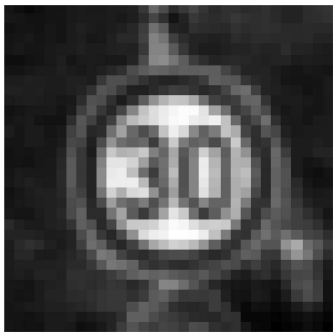
In [5]:

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

from sklearn.utils import shuffle

def preprocess(data):

    #Make the image grayscale
    image_p = np.mean(data, axis=3)

    #Expand dimensions
    image_p = np.expand_dims(image_p, axis=3)

    #Normalize
    image_p = (image_p - image_p.mean())/image_p.std()

    return image_p


train_f_p = preprocess(train_f)
valid_f_p = preprocess(valid_f)
test_f_p = preprocess(test_f)

print("Original shape:", train_f.shape)
print("Preprocessed shape:", train_f_p.shape)
fig, axs = plt.subplots(1,2, figsize=(10, 3))
axs = axs.ravel()

axs[0].axis('off')
axs[0].set_title('Original image sample:')
axs[0].imshow(train_f[4000].squeeze(), cmap='gray')

axs[1].axis('off')
axs[1].set_title('Preprocessed image sample:')
axs[1].imshow(train_f_p[4000].squeeze(), cmap='gray')
```

```
Original shape: (34799, 32, 32, 3)
Preprocessed shape: (34799, 32, 32, 1)

Out[5]:

<matplotlib.image.AxesImage at 0xcfe0390>
```

Original image sample:

Preprocessed image sample:



Model Architecture (Based on the original LeNet model architecture)

In [6]:

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

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

def LeNet(x):

    # Hyperparameters
    mu = 0
    sigma = 0.1

    print("Layer 0 shape:",x.get_shape())

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    W1 = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma
))
    b1 = tf.Variable(tf.zeros(6))
    x = tf.nn.bias_add(tf.nn.conv2d(x, W1, strides=[1, 1, 1, 1], padding='VALID'), b1)

    print("Layer 1 shape:",x.get_shape())

    # Activation.
    x = tf.nn.relu(x)

    # Dropout
    x = tf.nn.dropout(x, keep_prob1)

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

    # Layer 2: Convolutional. Output = 10x10x16.
    W2 = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigma
))
    b2 = tf.Variable(tf.zeros(16))
    x = tf.nn.bias_add(tf.nn.conv2d(x, W2, strides=[1, 1, 1, 1], padding='VALID'), b2)

    print("Layer 2 shape:",x.get_shape())

    # Activation.
    x = tf.nn.relu(x)

    # Pooling. Input = 10x10x16. Output = 5x5x16.
    x = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # Flatten. Input = 5x5x16. Output = 400.
    x = flatten(x)

    # Layer 3: Fully Connected. Input = 400. Output = 120.
    W3 = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
    b3 = tf.Variable(tf.zeros(120))
    x = tf.add(tf.matmul(x, W3), b3)

    print("Layer 3 shape:",x.get_shape())

    # Activation.
    x = tf.nn.relu(x)

    # Dropout
```

```
    x = tf.nn.dropout(x, keep_prob2)

    # Layer 4: Fully Connected. Input = 120. Output = 84.
    W4 = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
    b4 = tf.Variable(tf.zeros(84))
    x = tf.add(tf.matmul(x, W4), b4)

    # Activation.
    x = tf.nn.relu(x)

    # Dropout
    x = tf.nn.dropout(x, keep_prob3)

    # Layer 5: Fully Connected. Input = 84. Output = 43 (Number of classes).
    W5 = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stddev = sigma))
    b5 = tf.Variable(tf.zeros(43))
    logits = tf.add(tf.matmul(x, W5), b5)

    print("logits shape:",logits.get_shape())

    return logits
```

```
C:\Users\uidp7920\AppData\Local\Continuum\anaconda3\lib\site-packages\h5py
\__init__.py:36: FutureWarning: Conversion of the second argument of issub
dtype from `float` to `np.floating` is deprecated. In future, it will be t
reated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

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 [7]:

```
### 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.

tf.reset_default_graph()
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
keep_prob1 = tf.placeholder(tf.float32)
keep_prob2 = tf.placeholder(tf.float32)
keep_prob3 = tf.placeholder(tf.float32)
one_hot_y = tf.one_hot(y, classes_size)

LEARNING_RATE = .001
EPOCHS = 50
BATCH_SIZE = 128

def evaluate(data_f, data_l):
    num_examples = len(data_f)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_f, batch_l = data_f[offset:offset+BATCH_SIZE], data_l[offset:offset+BATCH
_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_f, y: batch_l, keep
_prob1:1.0, keep_prob2:1.0, keep_prob3:1.0})
        total_accuracy += (accuracy * len(batch_f))
    return total_accuracy / num_examples

logits = LeNet(x)
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 = LEARNING_RATE)
training_operation = optimizer.minimize(loss_operation)

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

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(train_f_p)

    print("Training has started...")
    print()
    validation_accuracy_figure = []

    for i in range(EPOCHS):
        train_f_p, train_l = shuffle(train_f_p, train_l)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_f, batch_l = train_f_p[offset:end], train_l[offset:end]
            sess.run(training_operation, feed_dict={x: batch_f, y: batch_l, keep_prob1:
0.7, keep_prob2:0.7, keep_prob3:0.6})

        validation_accuracy = evaluate(valid_f_p, valid_l)
```

```
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

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

```
Layer 0 shape: (?, 32, 32, 1)
Layer 1 shape: (?, 28, 28, 6)
Layer 2 shape: (?, 10, 10, 16)
Layer 3 shape: (?, 120)
logits shape: (?, 43)
Training has started...

EPOCH 1 ...
Validation Accuracy = 0.705

EPOCH 2 ...
Validation Accuracy = 0.845

EPOCH 3 ...
Validation Accuracy = 0.895

EPOCH 4 ...
Validation Accuracy = 0.909

EPOCH 5 ...
Validation Accuracy = 0.921

EPOCH 6 ...
Validation Accuracy = 0.931

EPOCH 7 ...
Validation Accuracy = 0.937

EPOCH 8 ...
Validation Accuracy = 0.940

EPOCH 9 ...
Validation Accuracy = 0.942

EPOCH 10 ...
Validation Accuracy = 0.942

EPOCH 11 ...
Validation Accuracy = 0.946

EPOCH 12 ...
Validation Accuracy = 0.948

EPOCH 13 ...
Validation Accuracy = 0.954

EPOCH 14 ...
Validation Accuracy = 0.946

EPOCH 15 ...
Validation Accuracy = 0.946

EPOCH 16 ...
Validation Accuracy = 0.951

EPOCH 17 ...
Validation Accuracy = 0.949

EPOCH 18 ...
Validation Accuracy = 0.953
```

```
EPOCH 19 ...
Validation Accuracy = 0.946

EPOCH 20 ...
Validation Accuracy = 0.950

EPOCH 21 ...
Validation Accuracy = 0.959

EPOCH 22 ...
Validation Accuracy = 0.955

EPOCH 23 ...
Validation Accuracy = 0.954

EPOCH 24 ...
Validation Accuracy = 0.952

EPOCH 25 ...
Validation Accuracy = 0.955

EPOCH 26 ...
Validation Accuracy = 0.958

EPOCH 27 ...
Validation Accuracy = 0.959

EPOCH 28 ...
Validation Accuracy = 0.954

EPOCH 29 ...
Validation Accuracy = 0.961

EPOCH 30 ...
Validation Accuracy = 0.961

EPOCH 31 ...
Validation Accuracy = 0.957

EPOCH 32 ...
Validation Accuracy = 0.953

EPOCH 33 ...
Validation Accuracy = 0.963

EPOCH 34 ...
Validation Accuracy = 0.957

EPOCH 35 ...
Validation Accuracy = 0.956

EPOCH 36 ...
Validation Accuracy = 0.964

EPOCH 37 ...
Validation Accuracy = 0.969

EPOCH 38 ...
Validation Accuracy = 0.964

EPOCH 39 ...
```

```
Validation Accuracy = 0.956

EPOCH 40 ...
Validation Accuracy = 0.965

EPOCH 41 ...
Validation Accuracy = 0.960

EPOCH 42 ...
Validation Accuracy = 0.966

EPOCH 43 ...
Validation Accuracy = 0.960

EPOCH 44 ...
Validation Accuracy = 0.956

EPOCH 45 ...
Validation Accuracy = 0.959

EPOCH 46 ...
Validation Accuracy = 0.951

EPOCH 47 ...
Validation Accuracy = 0.959

EPOCH 48 ...
Validation Accuracy = 0.964

EPOCH 49 ...
Validation Accuracy = 0.963

EPOCH 50 ...
Validation Accuracy = 0.958

Model saved
```

In [8]:

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

    valid_accuracy = evaluate(valid_f_p, valid_l)
    print("Valididation Accuracy = {:.3f}".format(valid_accuracy))

    test_accuracy = evaluate(test_f_p, test_l)
    print("Testing Accuracy = {:.3f}".format(test_accuracy))
```

```
INFO:tensorflow:Restoring parameters from .\Model
Valididation Accuracy = 0.958
Testing Accuracy = 0.943
```

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.

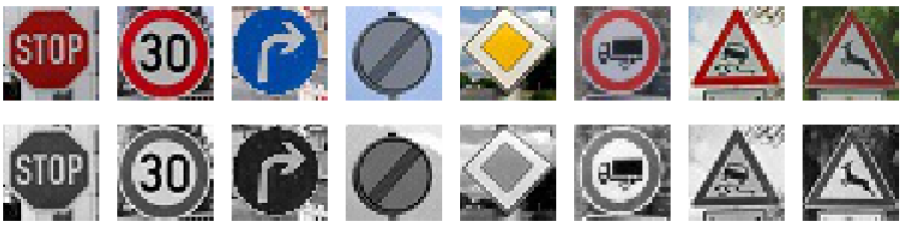
You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

Load and Output the Images

In [9]:

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

import os
import csv
import glob
import matplotlib.image as mpimg
import matplotlib.pyplot as plt


web_images = []

fig, axs = plt.subplots(2,8, figsize=(8, 2))
axs = axs.ravel()

for i, image_name in enumerate(glob.glob('./From-the-web/*.jpg')):
    print(image_name)
    image = cv2.imread(image_name)
    axs[i].axis('off')
    axs[i].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    web_images.append(image)

web_images_p =preprocess(web_images)
for i in range(len(web_images_p)):
    axs[i+8].axis('off')
    axs[i+8].imshow(web_images_p[i].squeeze(), cmap='gray')


test_labels = np.array([
        14,  # Stop
        1,  # Speed limit (30km/h)
        33,  # Turn right ahead
        32,  # End of all speed and passing limits
        12,  # Priority road
        16,  # Vehicles over 3.5 metric tons prohibited
        23,  # Slippery road
        31,  # Wild animals crossing
    ])

print(web_images_p.shape)
```

```
./From-the-web\01.jpg
./From-the-web\02.jpg
./From-the-web\03.jpg
./From-the-web\04.jpg
./From-the-web\05.jpg
./From-the-web\06.jpg
./From-the-web\07.jpg
./From-the-web\08.jpg
(8, 32, 32, 1)
```



Predict the Sign Type for Each Image

In [10]:

```
### 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 earl
ier.
### Feel free to use as many code cells as needed.


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver3 = tf.train.import_meta_graph('./Model.meta')
    saver3.restore(sess, "./Model")
    acc = evaluate(web_images_p, test_labels)
    print("Accuracy over new images = {:.3f}".format(acc))
```

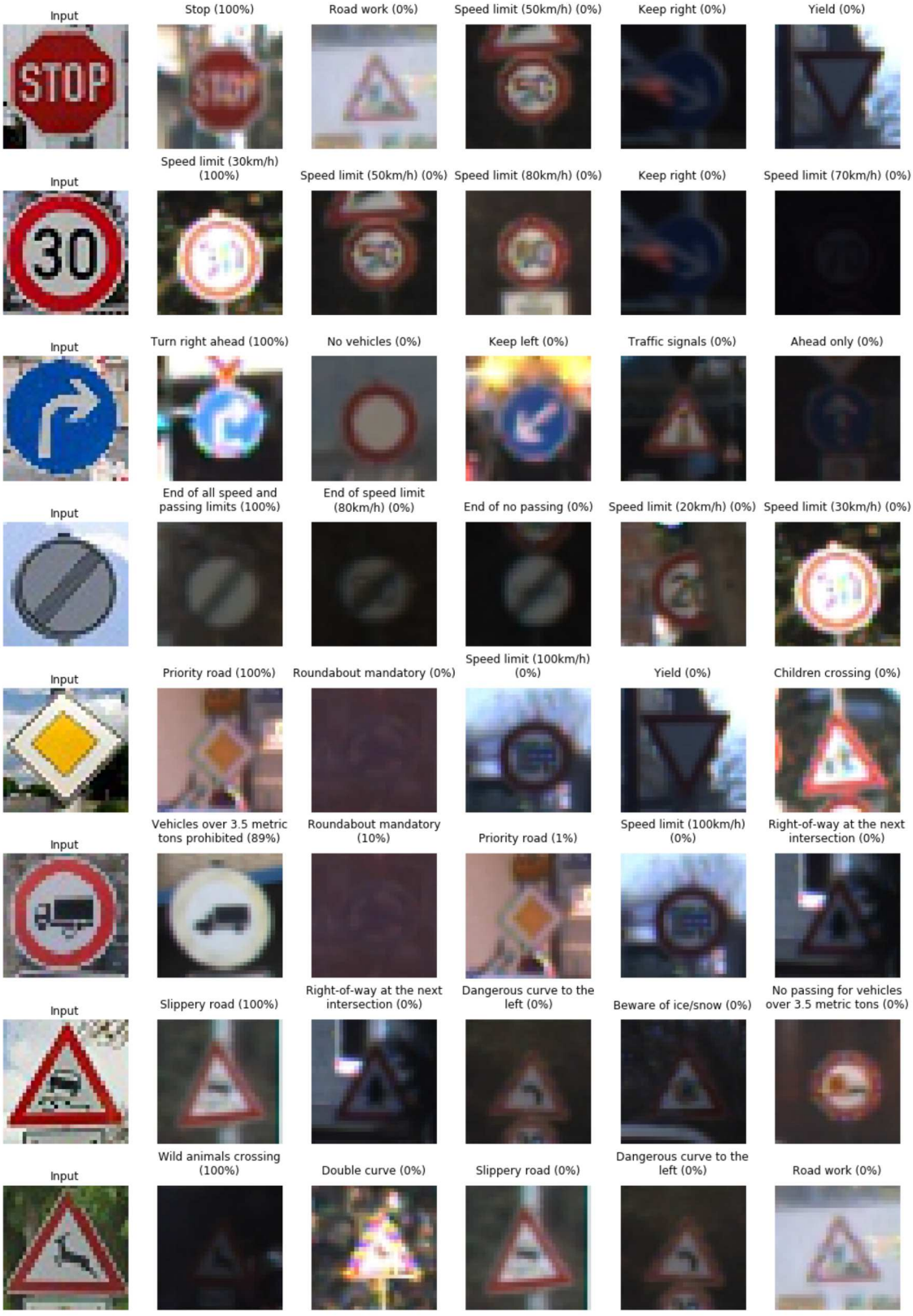
```
INFO:tensorflow:Restoring parameters from ./Model
Accuracy over new images = 1.000
```

Analyze Performance

In [11]:

```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate o
n these new images.

### Visualize the softmax probabilities here.
### Feel free to use as many code cells as needed.
from textwrap import wrap

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

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.import_meta_graph('./Model.meta')
    saver.restore(sess, "./Model")
    softmax_logits = sess.run(softmax_logits, feed_dict={x: web_images_p, keep_prob1:
1.0, keep_prob2:1.0, keep_prob3:1.0})
    p_result = sess.run(softmax_top_k, feed_dict={x: web_images_p, keep_prob1: 1.0, kee
p_prob2:1.0, keep_prob3:1.0})


    fig, axs = plt.subplots(len(web_images),(TOP_K+1), figsize=(14, 20))
    fig.subplots_adjust(top=0.8, hspace = 2.5, wspace=.5)
    fig.tight_layout()
    axs = axs.ravel()

    for i, image in enumerate(web_images):
        axs[(TOP_K+1)*i].axis('off')
        axs[(TOP_K+1)*i].imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        axs[(TOP_K+1)*i].set_title("Input")

        for j in range(TOP_K):
            p = p_result[1][i][j];
            p_percent = p_result[0][i][j] * 100
            index = np.argwhere(valid_l == p)[0]
            ax = axs[(TOP_K+1)*i+j+1]
            ax.axis('off')
            ax.imshow(valid_f[index].squeeze(), cmap='gray')
            title = ax.set_title("\n".join(wrap("{} ({:.0f}%)".format(signnames[p], p_p
ercent), 25)))
            title.set_y(1.05)
```

```
INFO:tensorflow:Restoring parameters from ./Model
```


Input
Stop (100%)
Road work (0%)
Speed limit (50km/h) (0%)
Keep right (0%)
Yield (0%)
Input
Speed limit (30km/h) (100%)
Speed limit (50km/h) (0%)
Speed limit (80km/h) (0%)
Keep right (0%)
Speed limit (70km/h) (0%)
Input
Turn right ahead (100%)
No vehicles (0%)
Keep left (0%)
Traffic signals (0%)
Ahead only (0%)
Input
End of all speed and passing limits (100%)
End of speed limit (80km/h) (0%)
End of no passing (0%)
Speed limit (20km/h) (0%)
Speed limit (30km/h) (0%)
Input
Priority road (100%)
Roundabout mandatory (0%)
Speed limit (100km/h) (0%)
Yield (0%)
Children crossing (0%)
Input
Vehicles over 3.5 metric tons prohibited (89%)
Roundabout mandatory (10%)
Priority road (1%)
Speed limit (100km/h) (0%)
Right-of-way at the next intersection (0%)
Input
Slippery road (100%)
Right-of-way at the next intersection (0%)
Dangerous curve to the left (0%)
Beware of ice/snow (0%)
No passing for vehicles over 3.5 metric tons (0%)
Input
Wild animals crossing (100%)
Double curve (0%)
Slippery road (0%)
Dangerous curve to the left (0%)
Road work (0%)
Input

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

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). `tf.nn.top_k` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k) could prove helpful here.

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

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

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `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:

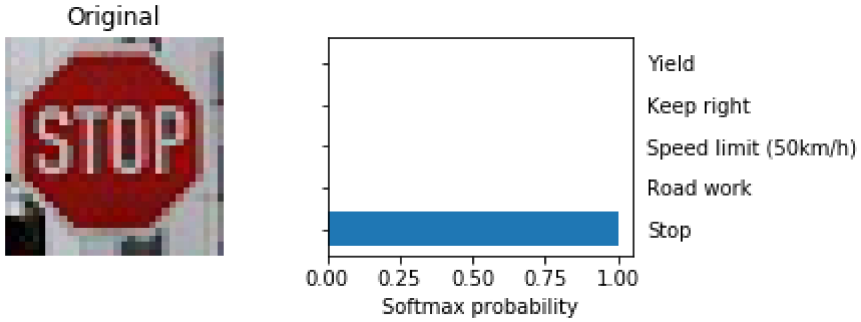
```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],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

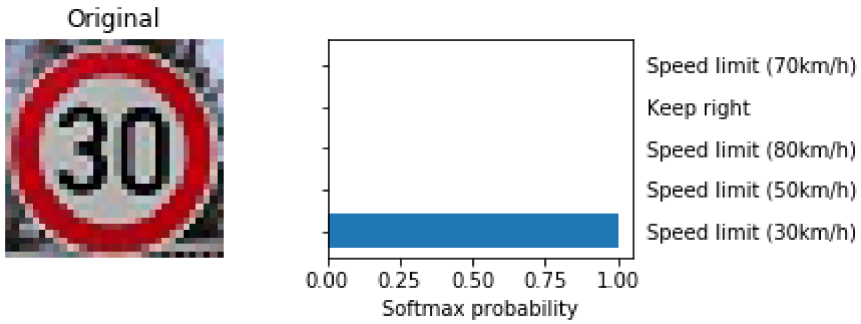
In [12]:

```
import matplotlib.gridspec as gridspec
from skimage import io
import os

for i in range(8):
    print("Actual label : ", signnames[test_labels[i]])

    # Prepare the grid
    plt.figure(figsize=(6, 2))
    gridspec.GridSpec(1, 2)

    # Plot original image
    plt.subplot2grid((1, 2), (0, 0), colspan=1, rowspan=1)
    plt.imshow(cv2.cvtColor(web_images[i], cv2.COLOR_BGR2RGB))
    plt.title("Original")
    plt.axis('off')

    # Plot predictions
    plt.subplot2grid((1, 2), (0, 1), colspan=1, rowspan=1)
    plt.barh(np.arange(5) + .5, p_result[0][i], align='center')
    plt.yticks(np.arange(5) + .5, signnames[p_result[1][i].astype(int)])
    plt.tick_params(axis='both', which='both', labelleft=False, labelright=True, labelt
op=False,
                    labelbottom=True)
    plt.xlabel('Softmax probability')
    plt.show()
```

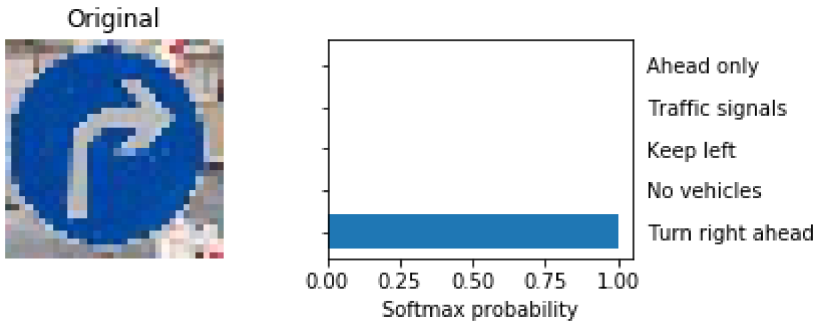
Actual label : Stop



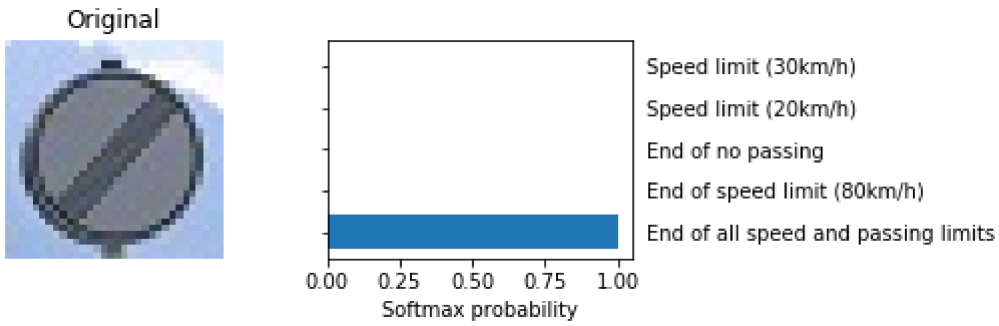
Actual label : Speed limit (30km/h)



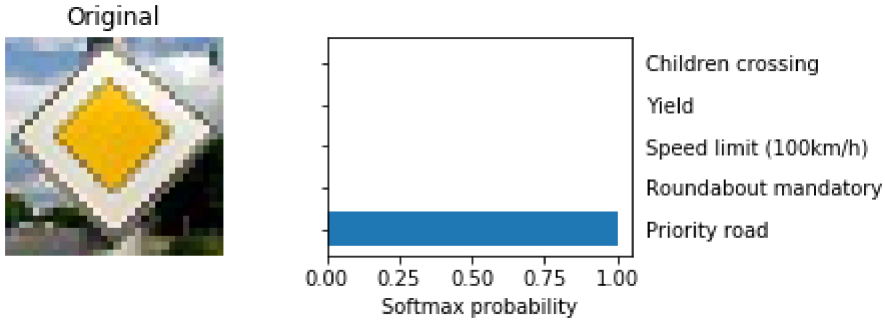
Actual label : Turn right ahead



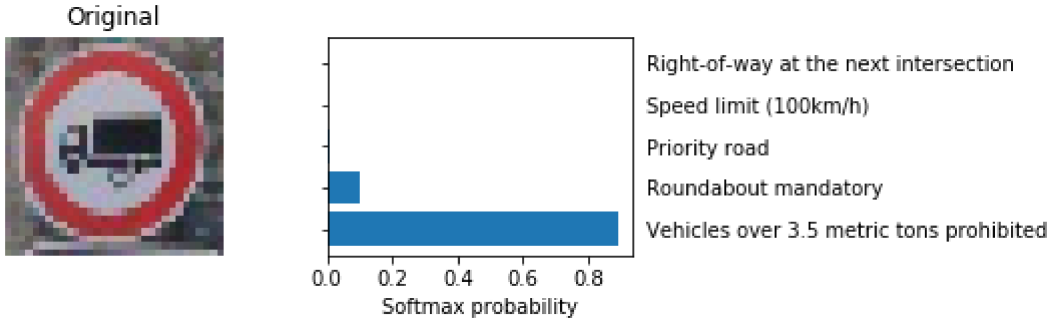
Actual label : End of all speed and passing limits



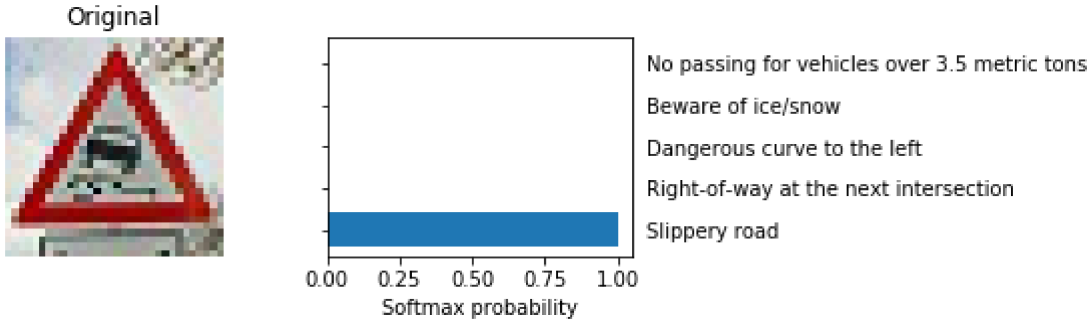
Actual label : Priority road



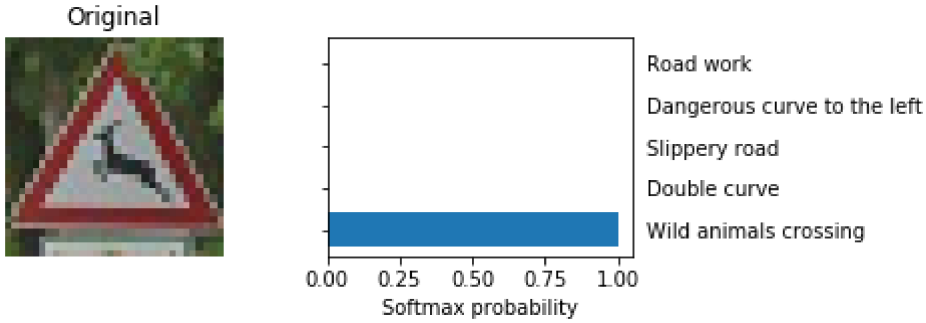

```
Actual label :  Vehicles over 3.5 metric tons prohibited
```




```
Actual label :  Slippery road
```




```
Actual label :  Wild animals crossing
```



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)](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.

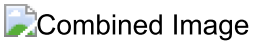
Note: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include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

In [13]:

```
### 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 de
fault 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 bloc
k, 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 var
iable
    # If you get an error tf_activation is not defined it may be having trouble accessi
ng 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 eac
h 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 =ac
tivation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax=act
ivation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin=act
ivation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap="gr
ay")
```