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]:
```
# Consolidated import Statements
import pickle
import numpy as np
from sklearn import preprocessing
import random
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from matplotlib.ticker import AutoMinorLocator
import tensorflow as tf
from sklearn.utils import shuffle
from scipy.ndimage import interpolation
import cv2
```

In [2]:
```
### Load pickled data

# Data path and file names for data load
data_path = '../DataFiles/TrafficSignData/'
training_file = ''.join([data_path, 'train.p'])
validation_file = ''.join([data_path, 'valid.p'])
testing_file = ''.join([data_path, '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 the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

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

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

In [3]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the r
esults

# Number of examples
n_train = len(y_train)
n_valid = len(y_valid)
n_test = len(y_test)

# What's the shape of an traffic sign image?
image_shape = X_train[0].shape

# How many unique classes/labels there are in the dataset.
n_classes = len(set(np.append(y_train, np.append(y_test, y_valid))))

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

Include an exploratory visualization of the dataset

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

The Matplotlib (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

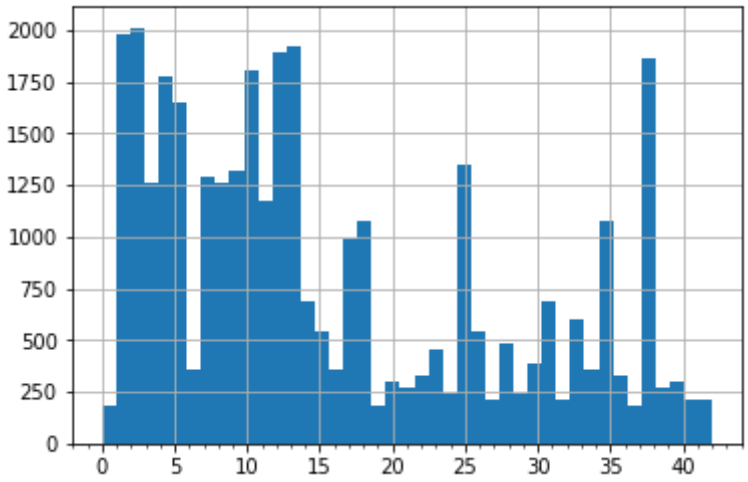
NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections.

In [4]:
```
### Print basic image helper
def printImage(img):
    cleaned_image = np.squeeze(img)
    plt.figure(figsize=(1,1))
    plt.imshow(cleaned_image)
```

In [5]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.

# Visualizations will be shown in the notebook.
%matplotlib inline

# Starting with basic random image from data set
index = random.randint(0, len(X_train))
printImage(X_train[index])
print(y_train[index])
```

```
30
```

In [6]:
```
# Display histogram of labels

def printHistogram(y, n_classes):
    fix, ax = plt.subplots()
    plt.hist(y, bins=n_classes)
    plt.xticks(range(0, n_classes, 5))
    ax.xaxis.set_minor_locator(AutoMinorLocator(5))
    ax.grid(True)
    plt.show()

printHistogram(y_train, n_classes)
```



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).

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

- Neural network architecture
- 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.

NOTE: 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!

Pre-process the Data Set (normalization, grayscale, etc.)

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

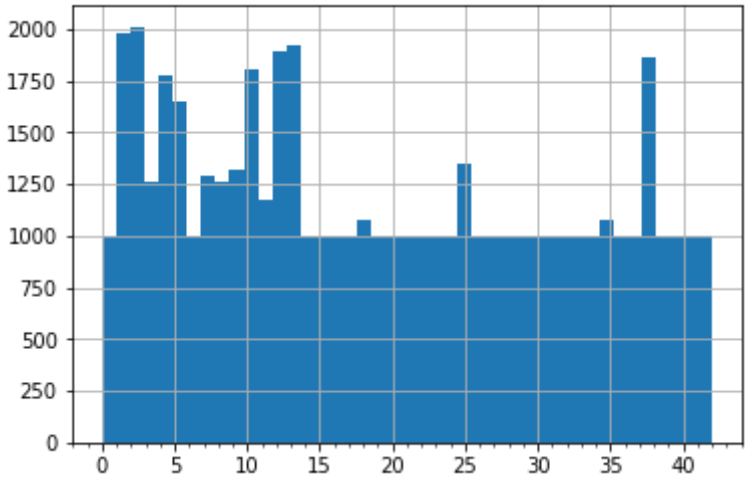
In [7]:
```
### Helper functions for conversion

def print_progress(current, total):
    tenths_complete = int((10 * current) // total)
    done = 'x' * tenths_complete
    not_done = '-' * (10 - tenths_complete)
    progress = done + not_done
    print("Working: {}  {} of {} complete".format(progress, current,
total), end='\r')

def normalize(x):
    # Convert to numpy
    x = np.array(x, dtype=np.float64)

    # Zero-center Data across each color channel
    x -= np.mean(x, axis=0)

    # Scale image data between -1 and 1 after centering over zero
    x = np.divide(x, 127.5)

    return x
def pcaWhiten(X_images):
    from sklearn.decomposition import PCA
    pca = PCA(copy=False, whiten=True)
    for x in X_images:
        for c in range(3):
            pca.fit_transform(x[...,c])

    return X_images

def equalHist(X_images):
    for x in X_images:
        for c in range(3):
            x[...,c] = cv2.equalizeHist(x[...,c])
    return X_images

def balanceClasses(X, y):
    print("Adding images to balance classes...")
    # Get counts for each class and largest count in the classes
    classCounts = np.bincount(y)
    highCount = 1000

    totalImages = np.sum(np.maximum(classCounts, 1000))
    currentImages = len(X)

    # Loop through each class and add new images to bring the total c
ount up to the highest
    for i in range(classCounts.size):
        currentCount = classCounts[i]
```

```
        imageClass = i
        while (currentCount < highCount):
            # get randomly altered image from random image from class
            newImage = randomizeImage(X[getRandomImageIndex(X, y, i)]
[0])
            X = np.append(X, [newImage], axis=0)
            y = np.append(y, i)
            currentCount += 1
            currentImages += 1
            print_progress(currentImages, totalImages)
    return X, y

def getRandomImageIndex(X, y, imageClass = None):
    if (imageClass is None):
        rndIndex = random.randint(0, len(X) - 1)
    else:
        classIndexes = np.argwhere(y == imageClass)
        rndIndex = classIndexes[random.randint(0, len(classIndexes) -
 1)]
    return rndIndex

def randomizeImage(x):
    rotationMin = -30.0
    rotationMax = 30.0
    shiftMin = -5.0
    shiftMax = 5.0

    # determine change type, randomly some will be rotated (0), some
 will
    # be translated (2), some will be both(1), evenly distributed
    changeType = random.randint(0, 2)

    new_x = x

    if changeType <= 1: # then rotate
        degrees = random.uniform(rotationMin, rotationMax)
        new_x = rotateMe(new_x, degrees)

    if changeType >= 1: # then shift
        a_factor = random.uniform(shiftMin, shiftMax)
        b_factor = random.uniform(shiftMin, shiftMax)
        new_x = shiftMe(new_x, a_factor, b_factor)

    return new_x


def rotateMe(x, degrees):
    return interpolation.rotate(x, degrees, reshape=False, mode='near
est')

def shiftMe(x, a_factor, b_factor):
    return interpolation.shift(x, [a_factor, b_factor, 0], mode='near
est', order=3)
```

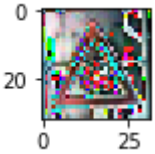
In [8]:
```
### Preprocessing Pipeline
def preprocess(X, y):

    X = equalHist(X)

    X = normalize(X)

    return X, y
```

In [9]:
```
### Preprocess the data here. Preprocessing steps could include normalization, converting to grayscale, etc.
### Feel free to use as many code cells as needed.

# Add supplemental training images to under-represented classes
print ("Starting sizes - x:{} y:{}".format(X_train.size,
y_train.size))
X_train, y_train = balanceClasses(X_train, y_train)
print ("Ending sizes - x:{} y:{}".format(X_train.size, y_train.size))
printHistogram(y_train, n_classes)

# Execute preprocessing chains
X_train, y_train = preprocess(X_train, y_train)
X_valid, y_valid = preprocess(X_valid, y_valid)
X_test, y_test = preprocess(X_test, y_test)

# Show preprocessing results example
print("Normalized values and image from previous random example:")
print(X_train[index])
printImage(X_train[index])

# Shuffle the data
X_train, y_train = shuffle(X_train, y_train)
```

```
Starting sizes - x:106902528 y:34799
Adding images to balance classes...
Ending sizes - x:158791680 y:516900 complete
```



```
Normalized values and image from previous random example:
[[[-0.51344053 -0.38521609 -0.30103156]
  [ 0.22944659  0.13781647  0.02625774]
  [ 0.30345051  0.19108547  0.02431964]
  ...,
  [-0.66624333 -0.5420013  -0.524222  ]
  [-0.95181136 -0.9061573  -0.8879184 ]
  [-1.00879633 -1.03483057 -0.99975222]]

 [[ 0.12423399  0.24263183  0.3961323 ]
  [ 0.91059157  0.8798821   0.92103983]
  [ 0.94235924  0.87421332  0.8862607 ]
  ...,
  [-0.62150315 -0.49975214 -0.51567512]
  [-0.94656592 -0.85738737 -0.84928006]
  [-1.00298051 -1.04755894 -1.00735956]]

 [[-0.16402733  0.20428452  0.40437996]
  [ 0.83773324  0.88391186  0.89352315]
  [ 0.91254287  0.88327822  0.92443762]
  ...,
  [-0.53451762 -0.46861933 -0.47548045]
  [-0.92894215 -0.84942254 -0.81737174]
  [-0.99318744 -1.03967514 -0.99997997]]


 ...,
 [[-0.687262   -0.47832076 -0.38584214]
  [-0.30342092 -0.0099052   0.14466332]
  [ 0.71585993  0.68120386  0.74219051]
  ...,
  [-0.16830426 -0.05127536 -0.05552407]
  [-0.47203563 -0.30157417 -0.32118914]
  [-0.66036075 -0.4912957  -0.4869564 ]]

 [[-0.66637412 -0.47280143 -0.37986685]
  [-0.3447401  -0.05896115  0.03278671]
  [ 0.5813238   0.68757639  0.70220128]
  ...,
  [-0.28593053 -0.09229198 -0.08065261]
  [-0.43330716 -0.27155008 -0.27546163]
  [-0.65404755 -0.48437176 -0.45675919]]

 [[-0.69240578 -0.49796031 -0.38865636]
  [-0.40872365 -0.1383923  -0.06231857]
  [ 0.51083844  0.61047284  0.60008785]
  ...,
  [-0.3262035  -0.14933294 -0.14619568]
  [-0.52837239 -0.3277524  -0.32465839]
  [-0.68703151 -0.57972301 -0.55351883]]]
```



Model Architecture

In [24]:
```
### Tweaked settings
EPOCHS = 30
BATCH_SIZE = 30
TRAINING_DROPOUT = 0.5
MU = 0
SIGMA = 0.1
LEARN_RATE = 0.001
```

In [25]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.

# this is used to easily experiment with different activation functio
ns
def my_activation(layer):
    return tf.nn.softsign(layer)

def LeNet(x):

    # Layer 1: Convolutional.  Input = 32x32x3.  Output = 28x28x12.
    l1_W = tf.Variable(tf.truncated_normal([5, 5, 3, 12], MU, SIGMA))
    l1_b = tf.Variable(tf.zeros(12))
    l1_strides = [1, 1, 1, 1]
    l1_padding = 'VALID'

    layer1 = tf.nn.conv2d(x, l1_W, l1_strides, l1_padding)
    layer1 = tf.nn.bias_add(layer1, l1_b)

    # Layer 1: Activation.
    layer1 = my_activation(layer1)

    # Layer 1: Pooling.  Input = 28x28x12.  Output = 14x14x12.
    p1_ksize = [1, 2, 2, 1]
    p1_strides = [1, 2, 2, 1]
    p1_padding = 'VALID'

    pooled1 = tf.nn.max_pool(layer1, p1_ksize, p1_strides,
p1_padding)

    # Layer 2: Convolutional. Input = 14x14x12. Output = 10x10x32
    l2_W = tf.Variable(tf.truncated_normal([5, 5, 12, 32], MU,
SIGMA))
    l2_b = tf.Variable(tf.zeros(32))
    l2_strides = [1, 1, 1, 1]
    l2_padding = 'VALID'

    layer2 = tf.nn.conv2d(pooled1, l2_W, l2_strides, l2_padding)
    layer2 = tf.nn.bias_add(layer2, l2_b)

    # Layer 2: Activiation.
    layer2 = my_activation(layer2)

    # Layer 2: Pooling.  Input = 10x10x32.  Output = 5x5x32.
```

```
    p2_ksize = [1, 2, 2, 1]
    p2_strides = [1, 2, 2, 1]
    p2_padding = 'VALID'

    pooled2 = tf.nn.max_pool(layer2, p2_ksize, p2_strides,
p2_padding)

    # Layer 3: Flatten.  Input = 5x5x32.  Output = 800.
    layer3 = tf.reshape(pooled2, [-1, 800])

    # Layer 3: Fully Connected.  Input = 800.  Output = 400.
    l3_W = tf.Variable(tf.truncated_normal([800, 400], MU, SIGMA))
    l3_b = tf.Variable(tf.zeros([400]))
    layer3 = tf.add(tf.matmul(layer3, l3_W), l3_b)

    # Layer 3: Activation.
    layer3 = my_activation(layer3)

    # Layer 3: Dropout
    layer3 = tf.nn.dropout(layer3, keep_prob)

    # Layer 3a: Fully Connected.  Input = 400.  Output = 120.
    l3a_W = tf.Variable(tf.truncated_normal([400, 120], MU, SIGMA))
    l3a_b = tf.Variable(tf.zeros([120]))
    layer3a = tf.add(tf.matmul(layer3, l3a_W), l3a_b)

    # Layer 3a: Activation.
    layer3a = my_activation(layer3a)

    # Layer 3a: Dropout
    layer3a = tf.nn.dropout(layer3a, keep_prob)

    # Layer 4: Fully connected.  Input = 120.  Output = 84.
    l4_W = tf.Variable(tf.truncated_normal([120, 84], MU, SIGMA))
    l4_b = tf.Variable(tf.zeros([84]))
    layer4 = tf.add(tf.matmul(layer3a, l4_W), l4_b)

    # Layer 4: Activation.
    layer4 = my_activation(layer4)

    # Layer 5: Final, fully connected.  Input = 84.  Output = 43.
    l5_W = tf.Variable(tf.truncated_normal([84, 43], MU, SIGMA))
    l5_b = tf.Variable(tf.zeros([43]))
    logits = tf.add(tf.matmul(layer4, l5_W), l5_b)

    return logits
```

Set up Features and Labels

In [26]:
```
x = tf.placeholder(tf.float32, (None, 32, 32, 3))
y = tf.placeholder(tf.int32, (None))
keep_prob = tf.placeholder(tf.float32)
one_hot_y = tf.one_hot(y, 43)
```

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.

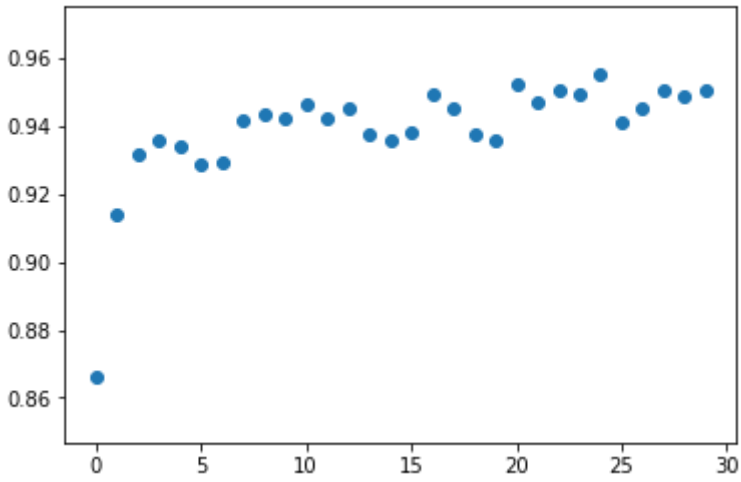
In [27]:
```
### Pipline setup
logits = LeNet(x)
cross_entropy =
tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=one_hot
_y)
loss_operation = tf.reduce_mean(cross_entropy)
#optimizer = tf.train.AdamOptimizer(learning_rate = LEARN_RATE)
optimizer = tf.train.AdamOptimizer(learning_rate = LEARN_RATE)
training_operation = optimizer.minimize(loss_operation)
```

In [28]:
```
### 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.fl
oat32))
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[o
ffset:offset+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 [29]:
```
### 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.

# Variable for collecting accuracy over epochs for graphing
epoch_accuracy = []

with tf.Session() as sess:

    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:en
d]
            sess.run(training_operation, feed_dict={x: batch_x, y: ba
tch_y, keep_prob: TRAINING_DROPOUT})

        validation_accuracy = evaluate(X_valid, y_valid)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accura
cy))
        print()
        epoch_accuracy.append(validation_accuracy)

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

plt.scatter(range(EPOCHS), epoch_accuracy)
plt.show()
```

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.866

EPOCH 2 ...
Validation Accuracy = 0.914

EPOCH 3 ...
Validation Accuracy = 0.932

EPOCH 4 ...
Validation Accuracy = 0.936

EPOCH 5 ...
Validation Accuracy = 0.934

EPOCH 6 ...
Validation Accuracy = 0.929

EPOCH 7 ...
Validation Accuracy = 0.929

EPOCH 8 ...
Validation Accuracy = 0.942

EPOCH 9 ...
Validation Accuracy = 0.944

EPOCH 10 ...
Validation Accuracy = 0.942

EPOCH 11 ...
Validation Accuracy = 0.947

EPOCH 12 ...
Validation Accuracy = 0.943

EPOCH 13 ...
Validation Accuracy = 0.945

EPOCH 14 ...
Validation Accuracy = 0.938

EPOCH 15 ...
Validation Accuracy = 0.936

EPOCH 16 ...
Validation Accuracy = 0.938

EPOCH 17 ...
Validation Accuracy = 0.950

EPOCH 18 ...
Validation Accuracy = 0.946

EPOCH 19 ...
```

```
Validation Accuracy = 0.938

EPOCH 20 ...
Validation Accuracy = 0.936

EPOCH 21 ...
Validation Accuracy = 0.952

EPOCH 22 ...
Validation Accuracy = 0.947

EPOCH 23 ...
Validation Accuracy = 0.951

EPOCH 24 ...
Validation Accuracy = 0.950

EPOCH 25 ...
Validation Accuracy = 0.956

EPOCH 26 ...
Validation Accuracy = 0.941

EPOCH 27 ...
Validation Accuracy = 0.946

EPOCH 28 ...
Validation Accuracy = 0.951

EPOCH 29 ...
Validation Accuracy = 0.949

EPOCH 30 ...
Validation Accuracy = 0.951

Model saved
```



In [30]:
```
# Accuracy Results
with tf.Session() as sess:
    saver.restore(sess, './signs')
    train_accuracy = evaluate(X_train, y_train)
    valid_accuracy = evaluate(X_valid, y_valid)
    test_accuracy = evaluate(X_test, y_test)
    print("  Final Training Accuracy =
{:.3f}".format(train_accuracy))
    print("Final Validation Accuracy =
{:.3f}".format(valid_accuracy))
    print("           Test Accuracy = {:.3f}".format(test_accuracy))
```

```
  Final Training Accuracy = 0.962
Final Validation Accuracy = 0.951
           Test Accuracy = 0.934
```

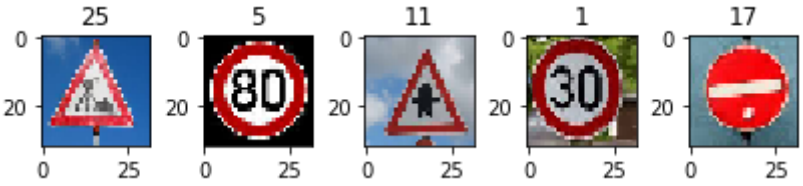
Step 3: Test a Model on New Images

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 [31]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.

image_path = './images/'
images = []
results = np.array([25, 5, 11, 1, 17], dtype=np.int32)

for i in range(0, 5):
    filename = 'sign' + str(i) + '-converted.bmp'
    images.append(mpimg.imread(image_path + filename))

# Print images
fig = plt.figure(figsize=(7, 1))
i = 0
for image in images:
    a = fig.add_subplot(1, 5, i + 1)
    plt.imshow(image)
    a.set_title(str(results[i]))
    i += 1
```



Predict the Sign Type for Each Image

In [32]:
```
### Run the predictions here and use the model to output the predicti
on 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.

# Preprocess
X_sample, y_sample = preprocess(images, results)

# Prediction function
predictions = tf.argmax(logits, 1)

# In new tf Session, restore variables and evaluate accurace
with tf.Session() as sess:
    saver.restore(sess, './signs')
    print("Model restored")
    sample_prediction = sess.run(predictions, feed_dict={x: X_sample,
 y: y_sample, keep_prob: 1.0})

# Display results
print('Prediction: ' + str(sample_prediction))
print('    Actual: ' + str(y_sample))
```

```
Model restored
Prediction: [29  0 28  0 17]
    Actual: [25  5 11  1 17]
```

Analyze Performance

In [33]:
```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, i
t's 20% accurate on these new images.
num_correct = np.sum(np.equal(sample_prediction, y_sample))
num_samples = np.size(sample_prediction)
print('  Accuracy: {:.1f}% ({} out of {} correct)'.format((float(num_
correct) / float(num_samples)) * 100,

num_correct, num_samples))
```

```
  Accuracy: 20.0% (1 out of 5 correct)
```

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

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

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

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

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tk.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.078934
97,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

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

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],
       [0, 5, 1],
       [1, 3, 5],
       [1, 4, 3]], dtype=int32))
```

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

In [34]:
```
### Print out the top five softmax probabilities for the predictions
 on the German traffic sign images found on the web.
### Feel free to use as many code cells as needed.
with tf.Session() as sess:
    saver.restore(sess, './signs')
    sm = tf.nn.softmax(logits)
    top5probabilities = sess.run(tf.nn.top_k(sm, k=5), feed_dict={x:
X_sample, y: y_sample, keep_prob: 1.0})


for i in range(5):
    print("Image{}:".format(i))
    for j in range(5):
        print('  Class: {} - Prob:
{:.3f}'.format(top5probabilities[1][i][j],
                                                top5probabilities[0]
[i][j]))
```

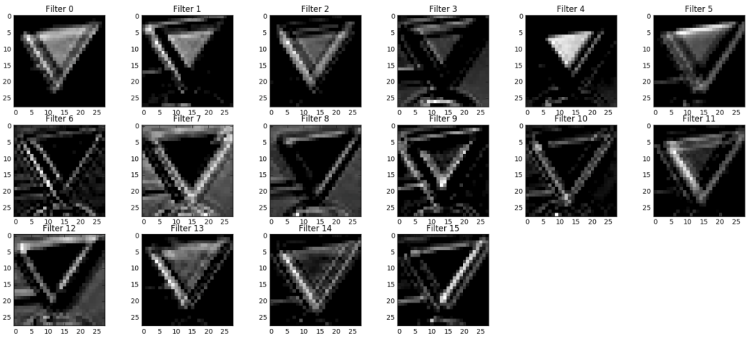
```
Image0:
  Class: 29 - Prob: 0.856
  Class: 24 - Prob: 0.062
  Class: 26 - Prob: 0.037
  Class: 22 - Prob: 0.014
  Class: 14 - Prob: 0.014
Image1:
  Class: 0 - Prob: 0.577
  Class: 4 - Prob: 0.130
  Class: 1 - Prob: 0.107
  Class: 8 - Prob: 0.088
  Class: 29 - Prob: 0.031
Image2:
  Class: 28 - Prob: 0.789
  Class: 30 - Prob: 0.158
  Class: 24 - Prob: 0.035
  Class: 29 - Prob: 0.014
  Class: 23 - Prob: 0.002
Image3:
  Class: 0 - Prob: 0.948
  Class: 40 - Prob: 0.022
  Class: 24 - Prob: 0.015
  Class: 1 - Prob: 0.005
  Class: 38 - Prob: 0.003
Image4:
  Class: 17 - Prob: 0.985
  Class: 14 - Prob: 0.015
  Class: 15 - Prob: 0.000
  Class: 26 - Prob: 0.000
  Class: 16 - Prob: 0.000
```

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

This Section is not required to complete but acts as an additional excersise for understaning the output of a neural network's weights. While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what it's feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image. From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the LeNet lab's (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) feature maps looked like for it's second convolutional layer you could enter conv2 as the tf_activation variable.

For an example of what feature map outputs look like, check out NVIDIA's results in their paper End-to-End Deep Learning for Self-Driving Cars (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/) in the section Visualization of internal CNN State. NVIDIA was able to show that their network's inner weights had high activations to road boundary lines by comparing feature maps from an image with a clear path to one without. Try experimenting with a similar test to show that your trained network's weights are looking for interesting features, whether it's looking at differences in feature maps from images with or without a sign, or even what feature maps look like in a trained network vs a completely untrained one on the same sign image.



Your output should look something like this (above)

In [35]:
```
### 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 t
he feature maps
# tf_activation: should be a tf variable name used during your traini
ng procedure that represents the calculated state of a specific weigh
t 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 a
nd max values of the output
# plt_num: used to plot out multiple different weight feature map set
s on the same block, just extend the plt number for each new feature
 map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1, a
ctivation_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your net
work expects
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow da
ta placeholder variable
    # If you get an error tf_activation is not defined it maybe havin
g 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 m
aps to show on each row and column
        plt.title('FeatureMap ' + str(featuremap)) # displays the fea
ture map number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap],
interpolation="nearest", vmin =activation_min, vmax=activation_max, c
map="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")
```

Question 9

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