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 pickle

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

training_file = "train.p"
validation_file= "valid.p"
testing_file = "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 (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]:

```
### 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 = X_train.shape[0]

# TODO: Number of validation examples
n_validation = X_test.shape[0]

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

# TODO: What's the shape of an traffic sign image?
image_shape = X_train.shape[1:]

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(np.unique(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
```

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
import pandas as pd
import random

# Visualizations will be shown in the notebook.
%matplotlib inline
sign_names = pd.read_csv('./signnames.csv').values

def plotImages(X, y, examples_per_sign=15, squeeze=False, cmap=None):
    samples_per_sign = np.bincount(y)
    for sign in sign_names:
        print("{0}. {1} - Samples: {2}".format(sign[0], sign[1], samples_per_sig
n[sign[0]]))
        sample_indices = np.where(y==sign[0])[0]
        random_samples = random.sample(list(sample_indices), examples_per_sign)
        fig = plt.figure(figsize = (examples_per_sign, 1))
        fig.subplots_adjust(hspace = 0, wspace = 0)
        for i in range(examples_per_sign):
            image = X[random_samples[i]]
            axis = fig.add_subplot(1,examples_per_sign, i+1, xticks=[], yticks=
[])
            if squeeze: image = image.squeeze()
            if cmap == None: axis.imshow(image)
            else: axis.imshow(image.squeeze(), cmap=cmap)
        plt.show()
```

Display the dataset images by sign type.

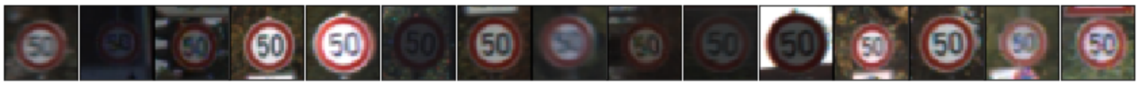
In [4]:

```
plotImages(X_train, y_train)
```

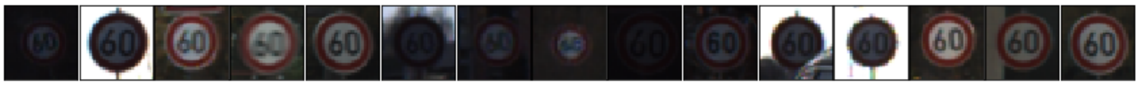
```
0. Speed limit (20km/h) - Samples: 180
```

```
1. Speed limit (30km/h) - Samples: 1980
```

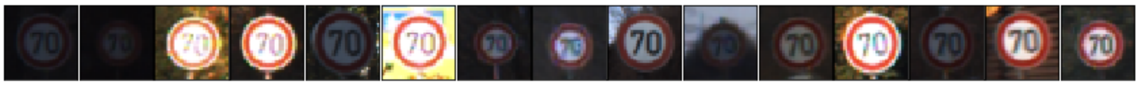
```
2. Speed limit (50km/h) - Samples: 2010
```



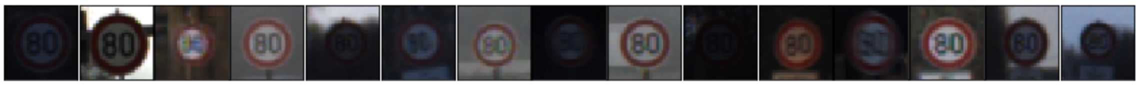
```
3. Speed limit (60km/h) - Samples: 1260
```



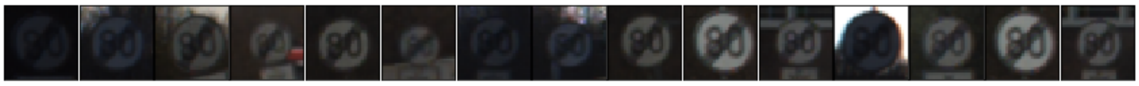
```
4. Speed limit (70km/h) - Samples: 1770
```



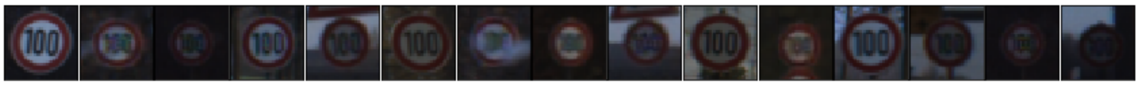
```
5. Speed limit (80km/h) - Samples: 1650
```



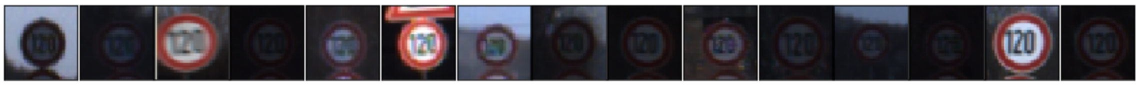
```
6. End of speed limit (80km/h) - Samples: 360
```



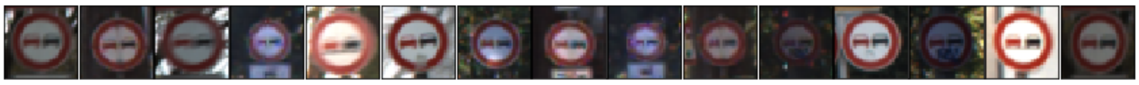
```
7. Speed limit (100km/h) - Samples: 1290
```



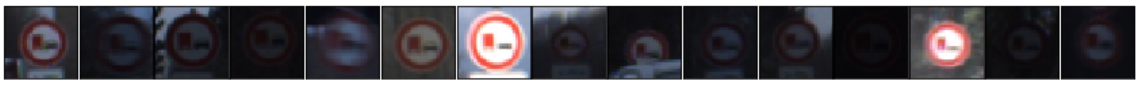
```
8. Speed limit (120km/h) - Samples: 1260
```



```
9. No passing - Samples: 1320
```

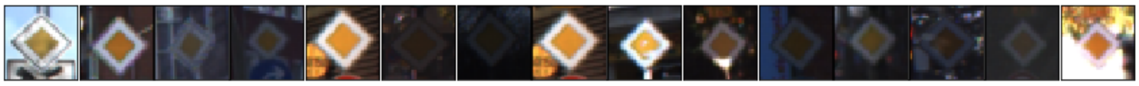


```
10. No passing for vehicles over 3.5 metric tons - Samples: 1800
```

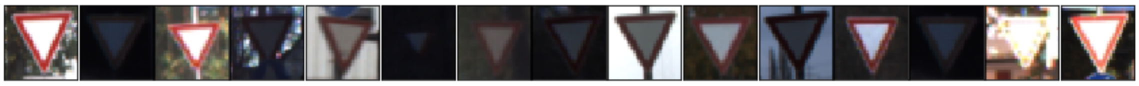


```
11. Right-of-way at the next intersection - Samples: 1170
```

```
12. Priority road - Samples: 1890
```

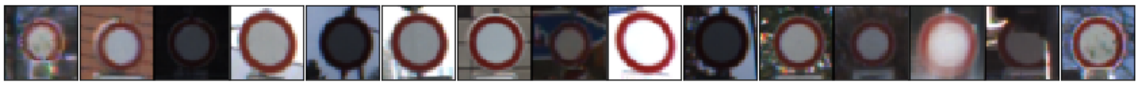


```
13. Yield - Samples: 1920
```

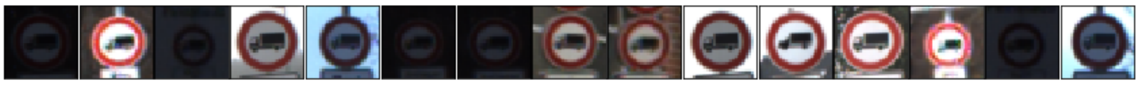


```
14. Stop - Samples: 690
```

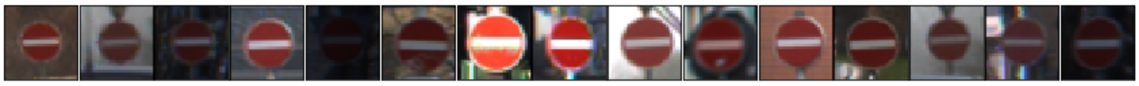
```
15. No vehicles - Samples: 540
```



```
16. Vehicles over 3.5 metric tons prohibited - Samples: 360
```



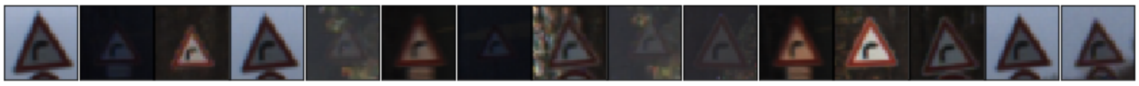
```
17. No entry - Samples: 990
```



```
18. General caution - Samples: 1080
```

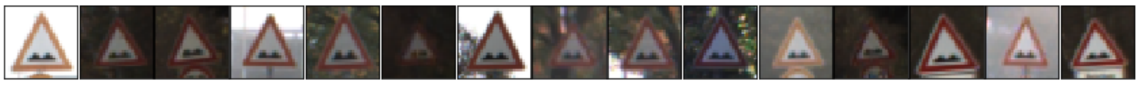
```
19. Dangerous curve to the left - Samples: 180
```

```
20. Dangerous curve to the right - Samples: 300
```



```
21. Double curve - Samples: 270
```

```
22. Bumpy road - Samples: 330
```

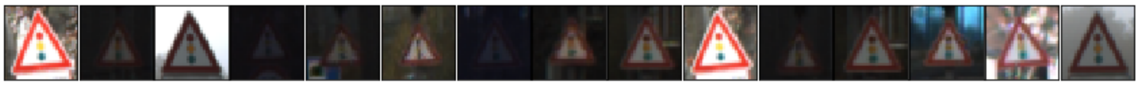


```
23. Slippery road - Samples: 450
```

```
24. Road narrows on the right - Samples: 240
```

```
25. Road work - Samples: 1350
```

```
26. Traffic signals - Samples: 540
```



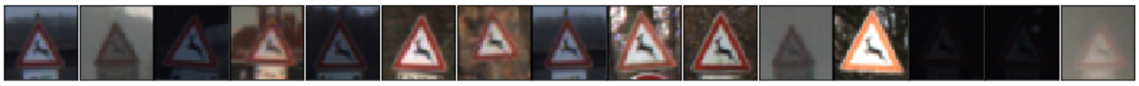
```
27. Pedestrians - Samples: 210
```

```
28. Children crossing - Samples: 480
```

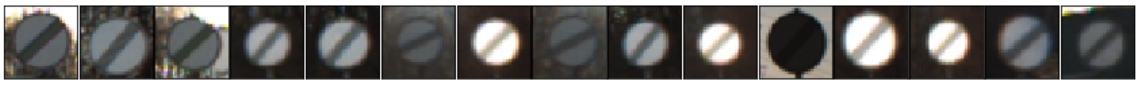
```
29. Bicycles crossing - Samples: 240
```

```
30. Beware of ice/snow - Samples: 390
```

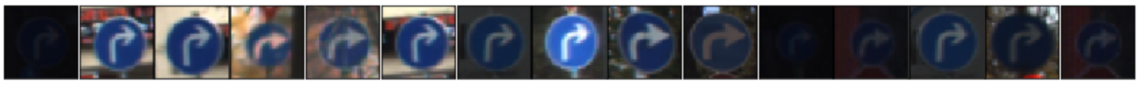
```
31. Wild animals crossing - Samples: 690
```



```
32. End of all speed and passing limits - Samples: 210
```



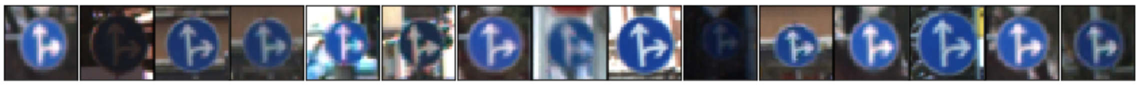
```
33. Turn right ahead - Samples: 599
```



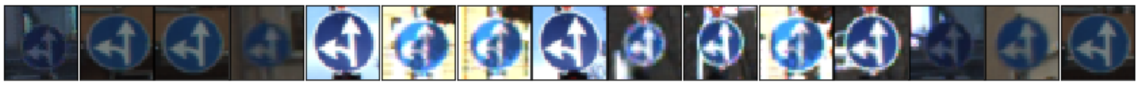
```
34. Turn left ahead - Samples: 360
```

```
35. Ahead only - Samples: 1080
```

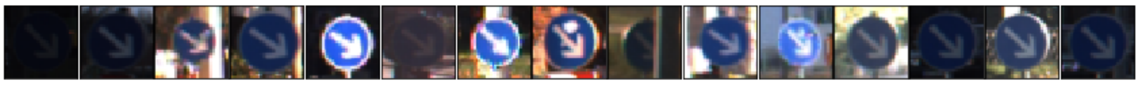
```
36. Go straight or right - Samples: 330
```



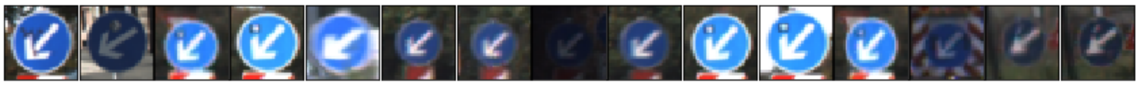
```
37. Go straight or left - Samples: 180
```



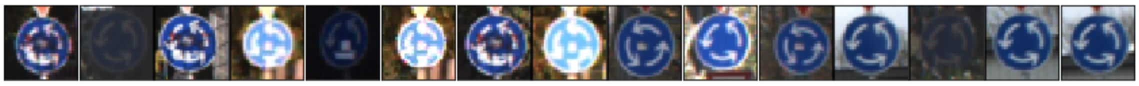
```
38. Keep right - Samples: 1860
```



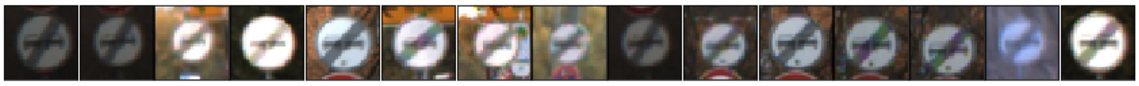
```
39. Keep left - Samples: 270
```



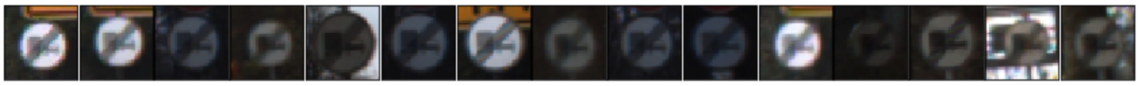
```
40. Roundabout mandatory - Samples: 300
```



```
41. End of no passing - Samples: 210
```



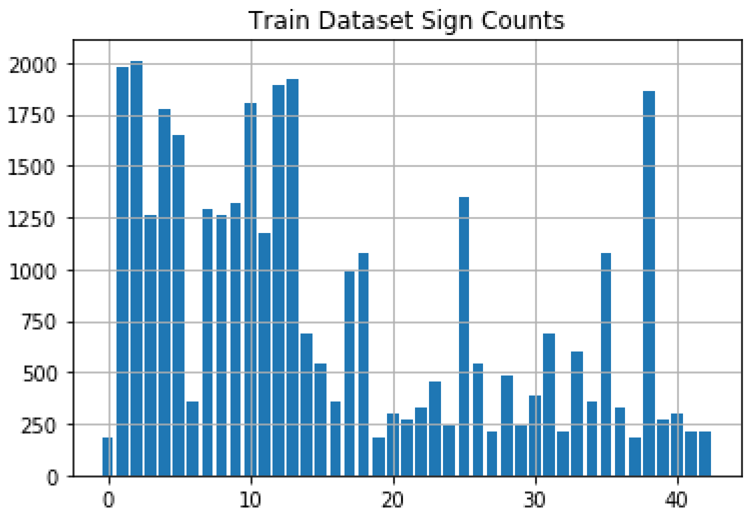
```
42. End of no passing by vehicles over 3.5 metric tons - Samples: 21
0
```



Let's compare the histagram of the different datasets, the distribution of the data appears to be uniform between the three datasets, but some signals have a lot less samples than others, this can indicate that we probably need to add fake data for those ones so we have a more uniform dataset.

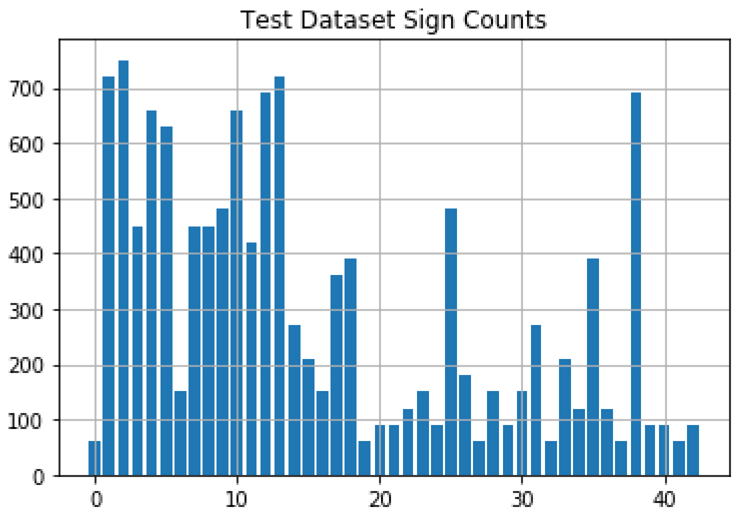
In [5]:

```
unique_train, counts_train = np.unique(y_train, return_counts=True)
plt.bar(unique_train, counts_train)
plt.grid()
plt.title("Train Dataset Sign Counts")
plt.show()
```



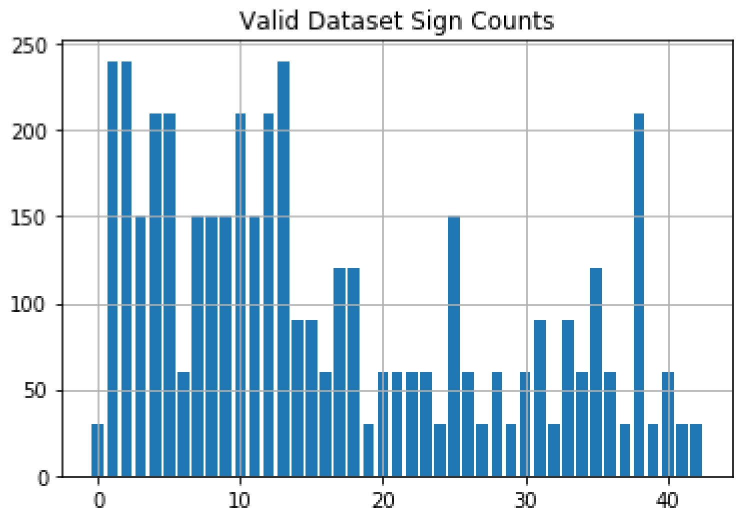
In [6]:

```
unique_test, counts_test = np.unique(y_test, return_counts=True)
plt.bar(unique_test, counts_test)
plt.grid()
plt.title("Test Dataset Sign Counts")
plt.show()
```



In [7]:

```
unique_valid, counts_valid = np.unique(y_valid, return_counts=True)
plt.bar(unique_valid, counts_valid)
plt.grid()
plt.title("Valid Dataset Sign Counts")
plt.show()
```



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 [8]:

```
### Preprocess the data here. Preprocessing steps could include normalization, c
onverting to grayscale, etc.
### Feel free to use as many code cells as needed.
def convertToGrayscale(images):
    return np.sum(images/3, axis=3, keepdims=True)

def preprocessImages(images):
    return (convertToGrayscale(images) - 128) / 128


X_train = preprocessImages(X_train)
X_valid = preprocessImages(X_valid)
X_test = preprocessImages(X_test)
```

In [9]:

```
plotImages(X_train, y_train, squeeze=True, cmap='gray')
```

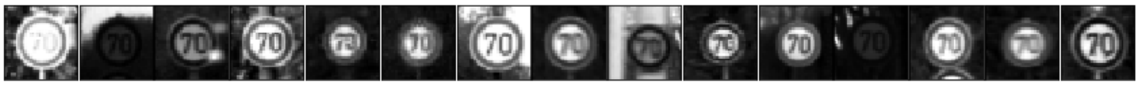
```
0. Speed limit (20km/h) - Samples: 180
```

```
1. Speed limit (30km/h) - Samples: 1980
```

```
2. Speed limit (50km/h) - Samples: 2010
```

```
3. Speed limit (60km/h) - Samples: 1260
```

```
4. Speed limit (70km/h) - Samples: 1770
```



```
5. Speed limit (80km/h) - Samples: 1650
```

```
6. End of speed limit (80km/h) - Samples: 360
```

```
7. Speed limit (100km/h) - Samples: 1290
```

```
8. Speed limit (120km/h) - Samples: 1260
```

```
9. No passing - Samples: 1320
```

```
10. No passing for vehicles over 3.5 metric tons - Samples: 1800
```

```
11. Right-of-way at the next intersection - Samples: 1170
```

12. Priority road - Samples: 1890

13. Yield - Samples: 1920

14. Stop - Samples: 690

15. No vehicles - Samples: 540

16. Vehicles over 3.5 metric tons prohibited - Samples: 360

17. No entry - Samples: 990

18. General caution - Samples: 1080

19. Dangerous curve to the left - Samples: 180

20. Dangerous curve to the right - Samples: 300

21. Double curve - Samples: 270

22. Bumpy road - Samples: 330

```
23. Slippery road - Samples: 450
```

```
24. Road narrows on the right - Samples: 240
```

```
25. Road work - Samples: 1350
```

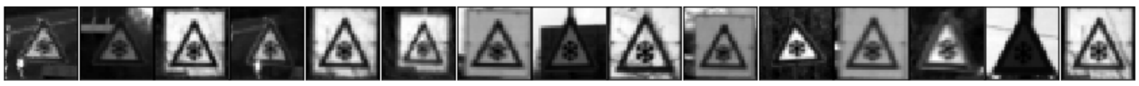
```
26. Traffic signals - Samples: 540
```

```
27. Pedestrians - Samples: 210
```

```
28. Children crossing - Samples: 480
```

```
29. Bicycles crossing - Samples: 240
```

```
30. Beware of ice/snow - Samples: 390
```

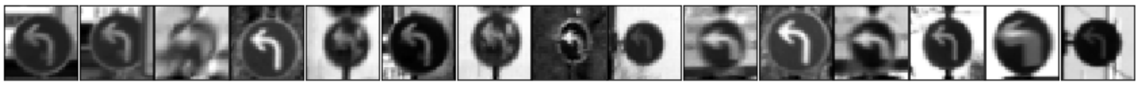


```
31. Wild animals crossing - Samples: 690
```

```
32. End of all speed and passing limits - Samples: 210
```

```
33. Turn right ahead - Samples: 599
```

```
34. Turn left ahead - Samples: 360
```



```
35. Ahead only - Samples: 1080
```

```
36. Go straight or right - Samples: 330
```

```
37. Go straight or left - Samples: 180
```

```
38. Keep right - Samples: 1860
```

```
39. Keep left - Samples: 270
```

```
40. Roundabout mandatory - Samples: 300
```

```
41. End of no passing - Samples: 210
```

```
42. End of no passing by vehicles over 3.5 metric tons - Samples: 21
0
```

Dataset augmentation

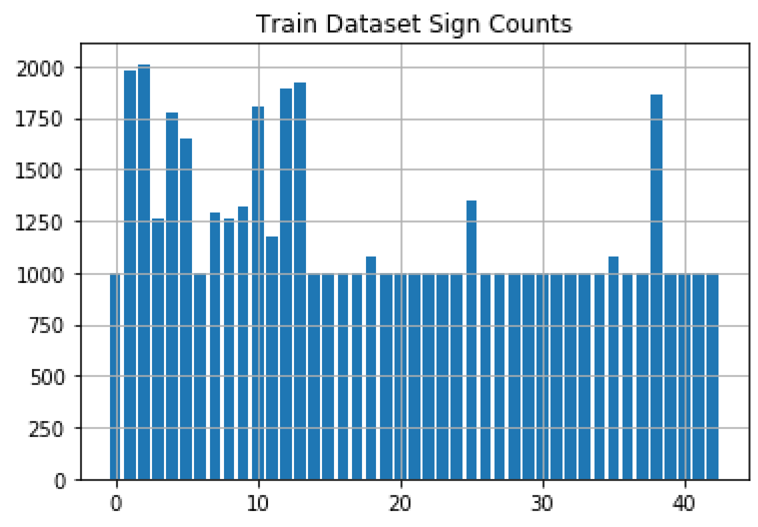
Due to the distribution of the dataset it needs to be augmented with modified images for labels that don't have enough values.

In [10]:

```
import cv2

number_classes = len(np.unique(y_train))
minimum_samples = 1000

#Transform functions
def image_translate(img):
    rows,cols,_ = img.shape
    px = 2
    dx, dy = np.random.randint(-px,px,2)
    M = np.float32([[1, 0, dx], [0, 1, dy]])
    result = cv2.warpAffine(img,M,(cols,rows))
    result = result[:,:,np.newaxis]

    return result

def image_scaling(img):
    rows,cols,_ = img.shape
    px = np.random.randint(-2,2)
    pts1 = np.float32([[px,px],[rows-px,px],[px,cols-px],[rows-px,cols-px]])
    pts2 = np.float32([[0,0],[rows,0],[0,cols],[rows,cols]])
    M = cv2.getPerspectiveTransform(pts1,pts2)
    result = cv2.warpPerspective(img,M,(rows,cols))
    result = result[:,:,np.newaxis]

    return result

def image_warped(img):
    rows,cols,_ = img.shape
    rndx = np.random.rand(3) - 0.5
    rndx *= cols * 0.06
    rndy = np.random.rand(3) - 0.5
    rndy *= rows * 0.06
    x1 = cols/4
    x2 = 3*cols/4
    y1 = rows/4
    y2 = 3*rows/4

    pts1 = np.float32([[y1,x1],
                       [y2,x1],
                       [y1,x2]])
    pts2 = np.float32([[y1+rndy[0],x1+rndx[0]],
                       [y2+rndy[1],x1+rndx[1]],
                       [y1+rndy[2],x2+rndx[2]]])

    M = cv2.getAffineTransform(pts1,pts2)
    result = cv2.warpAffine(img,M,(cols,rows))
    result = result[:,:,np.newaxis]

    return result

def image_brightness(img):
    shifted = img + 1.0   # shift to (0,2) range
    img_max_value = max(shifted.flatten())
    max_coef = 2.0/img_max_value
    min_coef = max_coef - 0.1
    coef = np.random.uniform(min_coef, max_coef)

    return shifted * coef - 1.0
```

```
new_X_train_images = []
new_Y_train_labels = []
for class_n in range(number_classes):
    class_indices = np.where(y_train == class_n)
    n_samples = len(class_indices[0])
    if n_samples < 1000:
        for i in range(1000 - n_samples):
            new_img = X_train[class_indices[0][i % n_samples]]
            new_img = image_translate(image_scaling(image_warped(image_brightnes
s(new_img))))
            new_X_train_images.append(new_img)
            new_Y_train_labels.append(class_n)

X_train = np.concatenate((X_train, new_X_train_images), axis=0)
y_train = np.concatenate((y_train, new_Y_train_labels), axis=0)

print("Updated trainining dataset.")
unique_train, counts_train = np.unique(y_train, return_counts=True)
plt.bar(unique_train, counts_train)
plt.grid()
plt.title("Train Dataset Sign Counts")
plt.show()
```

```
Updated trainining dataset.
```



Shuffle the training data

In [11]:

```
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
```

Model Architecture

Setup tensorflow

In [12]:

```
import tensorflow as tf

EPOCHS = 50
BATCH_SIZE = 100
```

```
/Users/feleir/miniconda3/envs/IntroToTensorFlow/lib/python3.6/site-p
ackages/h5py/__init__.py:36: FutureWarning: Conversion of the second
argument of issubdtype from `float` to `np.floating` is deprecated.
In future, it will be treated as `np.float64 == np.dtype(float).type
`.
  from ._conv import register_converters as _register_converters
```

Define LeNet architecture that accepts 32x32x1 images returning a fully connected logit of 43 outputs.

In [13]:

```
### Define your architecture here.
### Feel free to use as many code cells as needed.
from tensorflow.contrib.layers import flatten

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

    # Layer 1. Input = 32x32x1. Output = 28x28x48.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, std
dev = sigma))
    conv1_b = tf.Variable(tf.zeros([6]))
    conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID', name
='conv1') + conv1_b
    conv1 = tf.nn.relu(conv1, name='conv1_relu')

    # Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padd
ing='VALID', name='conv1_pool')

    # Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, st
ddev = sigma))
    conv2_b = tf.Variable(tf.zeros(16))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID'
, name='conv2') + conv2_b
    conv2   = tf.nn.relu(conv2, name='conv2_relu')

    # Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padd
ing='VALID', name='conv2_pool')

    # Layer 3: Convolutional. Output = 1x1x400.
    conv3_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 400), mean = mu,
stddev = sigma))
    conv3_b = tf.Variable(tf.zeros(400))
    conv3   = tf.nn.conv2d(conv2, conv3_W, strides=[1, 1, 1, 1], padding='VALID'
, name='conv3') + conv3_b
    conv3   = tf.nn.relu(conv3, name='conv3_relu')

    # Flatten, Output = 5x5x16 = 400
    fc0 = flatten(conv2)
    # Flatten, Output = 1x1x400 = 400
    fc1 = flatten(conv3)

    # Concat. Input = 400 + 400. Output = 800
    fcconcat = tf.concat([fc0, fc1], 1)
    fcconcat = tf.nn.dropout(fcconcat, keep_prob)

    # Layer 5. Fully connected
    # Input = 800 -> Output = 43 (Number of classes)
    fc3_w = tf.Variable(tf.truncated_normal(shape = (800, n_classes), mean = mu,
 stddev = sigma))
    fc3_b = tf.Variable(tf.zeros(n_classes))
    logits = tf.matmul(fcconcat, fc3_w) + fc3_b

    return logits
```

Train, Validate and Test the Model

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.

Training pipeline

In [14]:

```
# Features and labels
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
keep_prob = tf.placeholder(tf.float32)

rate = 0.0009

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 = rate)
training_operation = optimizer.minimize(loss_operation)
```

```
WARNING:tensorflow:From <ipython-input-14-4bb113b26401>:10: softmax_
cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is dep
recated and will be removed in a future version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See tf.nn.softmax_cross_entropy_with_logits_v2.
```

Model evaluation

In [15]:

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

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offse
t+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_
y, keep_prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

Train the model

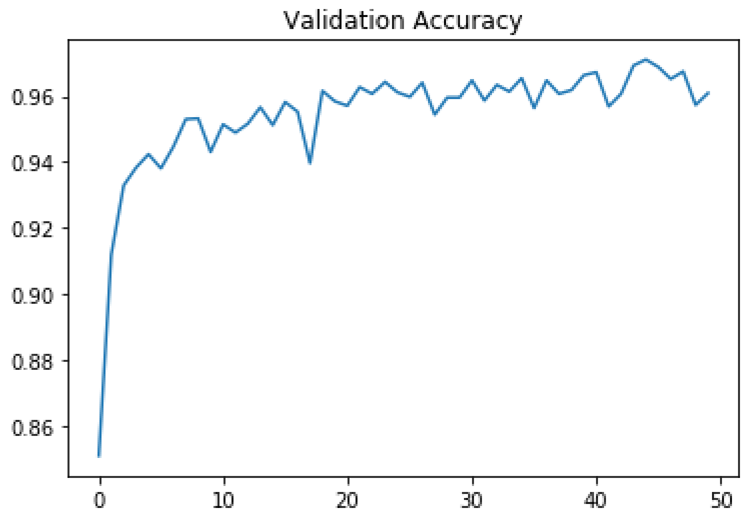
Run the training data through the training pipeline to train the model. Before each epoch, shuffle the training set. After each epoch, measure the loss and accuracy of the validation set. Save the model after training.

In [16]:

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

    print("Training...")
    print()
    validation_results = []
    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:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep
_prob: 0.5})

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

    plt.plot(validation_results)
    plt.title("Validation Accuracy")
    plt.show()

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

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.851
EPOCH 2 ...
Validation Accuracy = 0.912
EPOCH 3 ...
Validation Accuracy = 0.933
EPOCH 4 ...
Validation Accuracy = 0.938
EPOCH 5 ...
Validation Accuracy = 0.942
EPOCH 6 ...
Validation Accuracy = 0.938
EPOCH 7 ...
Validation Accuracy = 0.945
EPOCH 8 ...
Validation Accuracy = 0.953
EPOCH 9 ...
Validation Accuracy = 0.953
EPOCH 10 ...
Validation Accuracy = 0.943
EPOCH 11 ...
Validation Accuracy = 0.951
EPOCH 12 ...
Validation Accuracy = 0.949
EPOCH 13 ...
Validation Accuracy = 0.952
EPOCH 14 ...
Validation Accuracy = 0.957
EPOCH 15 ...
Validation Accuracy = 0.951
EPOCH 16 ...
Validation Accuracy = 0.958
EPOCH 17 ...
Validation Accuracy = 0.955
EPOCH 18 ...
Validation Accuracy = 0.940
EPOCH 19 ...
Validation Accuracy = 0.962
EPOCH 20 ...
Validation Accuracy = 0.959
EPOCH 21 ...
Validation Accuracy = 0.957
EPOCH 22 ...
Validation Accuracy = 0.963
EPOCH 23 ...
Validation Accuracy = 0.961
EPOCH 24 ...
Validation Accuracy = 0.964
EPOCH 25 ...
Validation Accuracy = 0.961
EPOCH 26 ...
Validation Accuracy = 0.960
EPOCH 27 ...
Validation Accuracy = 0.964
EPOCH 28 ...
Validation Accuracy = 0.954
EPOCH 29 ...
Validation Accuracy = 0.960
EPOCH 30 ...
```

```
Validation Accuracy = 0.960
EPOCH 31 ...
Validation Accuracy = 0.965
EPOCH 32 ...
Validation Accuracy = 0.959
EPOCH 33 ...
Validation Accuracy = 0.963
EPOCH 34 ...
Validation Accuracy = 0.961
EPOCH 35 ...
Validation Accuracy = 0.966
EPOCH 36 ...
Validation Accuracy = 0.956
EPOCH 37 ...
Validation Accuracy = 0.965
EPOCH 38 ...
Validation Accuracy = 0.961
EPOCH 39 ...
Validation Accuracy = 0.962
EPOCH 40 ...
Validation Accuracy = 0.966
EPOCH 41 ...
Validation Accuracy = 0.967
EPOCH 42 ...
Validation Accuracy = 0.957
EPOCH 43 ...
Validation Accuracy = 0.961
EPOCH 44 ...
Validation Accuracy = 0.969
EPOCH 45 ...
Validation Accuracy = 0.971
EPOCH 46 ...
Validation Accuracy = 0.969
EPOCH 47 ...
Validation Accuracy = 0.965
EPOCH 48 ...
Validation Accuracy = 0.968
EPOCH 49 ...
Validation Accuracy = 0.957
EPOCH 50 ...
Validation Accuracy = 0.961
```



```
Model saved
```

Test the model

Obtain model accuracy for the test dataset.

In [17]:

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

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

```
INFO:tensorflow:Restoring parameters from ./lenet
Test Accuracy = 0.958
```

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.

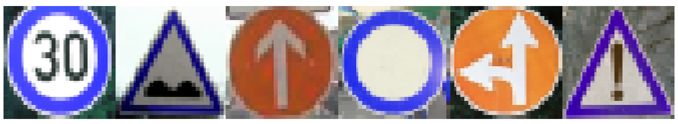
You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

Load and Output the Images

In [18]:

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

import glob
import matplotlib.image as mpimg

image_files = sorted(glob.glob('./my-images/*.png'))

fig, axs = plt.subplots(1,len(image_files))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

my_signs = []
my_labels = np.array([1, 22, 35, 15, 37, 18])

for i, img in enumerate(image_files):
    image = cv2.imread(img)
    axs[i].axis('off')
    axs[i].imshow(image.squeeze())
    my_signs.append(image)

my_signs = np.asarray(my_signs)
my_signs = preprocessImages(my_signs)

print(my_signs.shape)
```

```
(6, 32, 32, 1)
```



Predict the Sign Type for Each Image

In [19]:

```
### Run the predictions here and use the model to output the prediction for each
 image.
### Make sure to pre-process the images with the same pre-processing pipeline us
ed earlier.
### Feel free to use as many code cells as needed.
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    my_accuracy = evaluate(my_signs, my_labels)
    print("My Data Set Accuracy = {:.3f}".format(my_accuracy))
```

```
INFO:tensorflow:Restoring parameters from ./lenet
My Data Set Accuracy = 1.000
```

Analyze Performance

In [20]:

```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% acc
urate on these new images.
for i in range(6):
    with tf.Session() as sess:
        saver.restore(sess, tf.train.latest_checkpoint('.'))
        my_accuracy = evaluate([my_signs[i]], [my_labels[i]])
        print('Image {}'.format(i+1))
        print("Image Accuracy = {:.3f}".format(my_accuracy))
        print()
```

```
INFO:tensorflow:Restoring parameters from ./lenet
Image 1
Image Accuracy = 1.000

INFO:tensorflow:Restoring parameters from ./lenet
Image 2
Image Accuracy = 1.000

INFO:tensorflow:Restoring parameters from ./lenet
Image 3
Image Accuracy = 1.000

INFO:tensorflow:Restoring parameters from ./lenet
Image 4
Image Accuracy = 1.000

INFO:tensorflow:Restoring parameters from ./lenet
Image 5
Image Accuracy = 1.000

INFO:tensorflow:Restoring parameters from ./lenet
Image 6
Image Accuracy = 1.000
```

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,  0.0789
3497,
          0.12789202],
        [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
          0.15899337],
        [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
          0.23892179],
        [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
          0.16505091],
        [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
          0.09155967]])
```

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

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

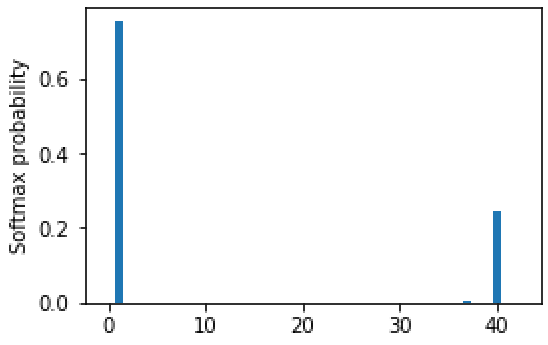
Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in `a`. You'll also notice `[3, 0, 5]` are the corresponding indices.

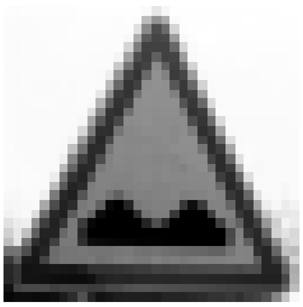
In [21]:

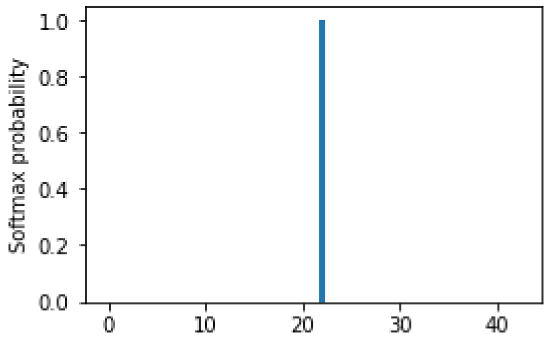
```
### Print out the top five softmax probabilities for the predictions on the Germ
an traffic sign images found on the web.
### Feel free to use as many code cells as needed.
softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k=3)


with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    my_softmax_logits = sess.run(softmax_logits, feed_dict={x: my_signs, keep_pr
ob: 1.0})
    my_top_k = sess.run(top_k, feed_dict={x: my_signs, keep_prob: 1.0})

fig, axs = plt.subplots(len(my_softmax_logits),2, figsize=(9, 19))
axs = axs.ravel()

for i in range(len(my_softmax_logits)*2):
    if i%2 == 0:
        axs[i].axis('off')
        axs[i].imshow(my_signs[i//2].squeeze(), cmap='gray')
    else:
        axs[i].bar(np.arange(n_classes), my_softmax_logits[(i-1)//2])
        axs[i].set_ylabel('Softmax probability')
```

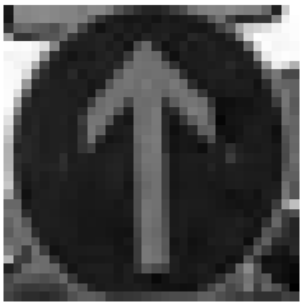
```
INFO:tensorflow:Restoring parameters from ./lenet
```

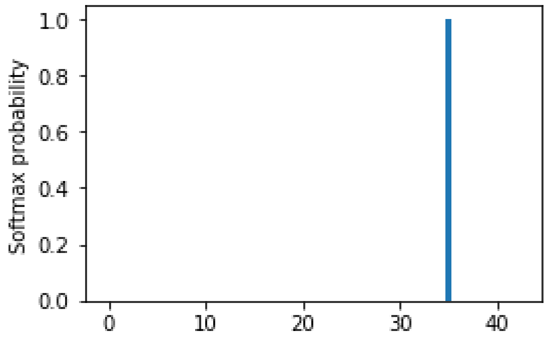
30

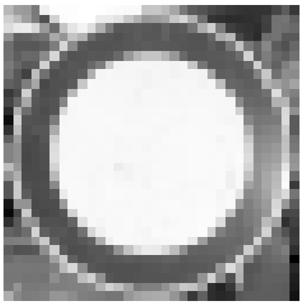

Softmax probability
0.0
0.2
0.4
0.6
0
10
20
30
40

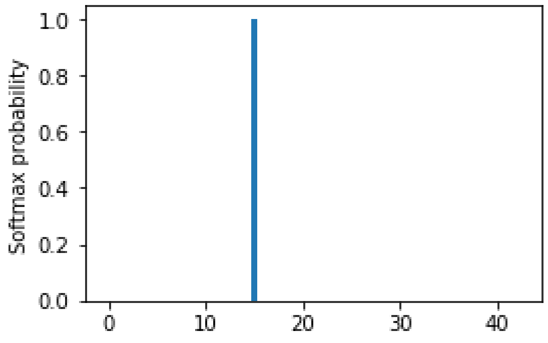


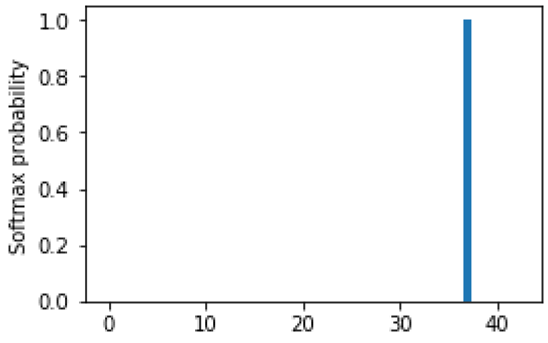

Softmax probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40

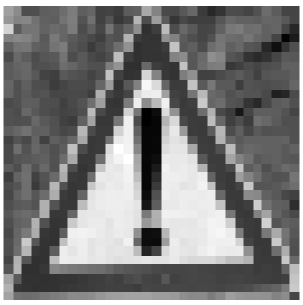


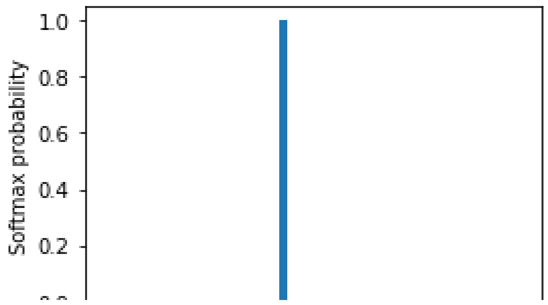

Softmax probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40

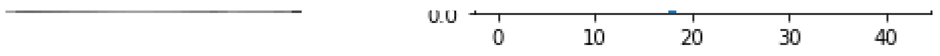



Softmax probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40


Softmax probability
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40




Softmax probability
0.2
0.4
0.6
0.8
1.0



Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) as a guide. The writeup can be in a markdown or pdf file.

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

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

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

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

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

Combined Image

Your output should look something like this (above)

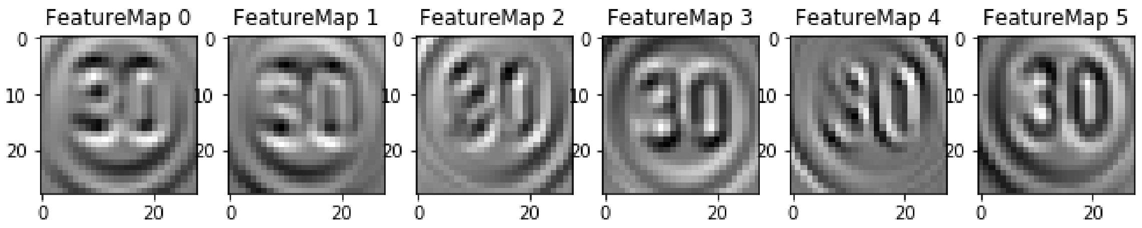
In [22]:

```
### 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 procedur
e that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more detai
l, by default matplot sets min and max to the actual min and max values of the o
utput
# plt_num: used to plot out multiple different weight feature map sets on the sa
me block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_m
ax=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network expect
s
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data placehol
der 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 nu
mber
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
min =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
max=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", v
min=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", c
map="gray")
```

In [23]:

```
# First Convolutional layer
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv1 = sess.graph.get_tensor_by_name('conv1:0')
    outputFeatureMap(my_signs, conv1)
```

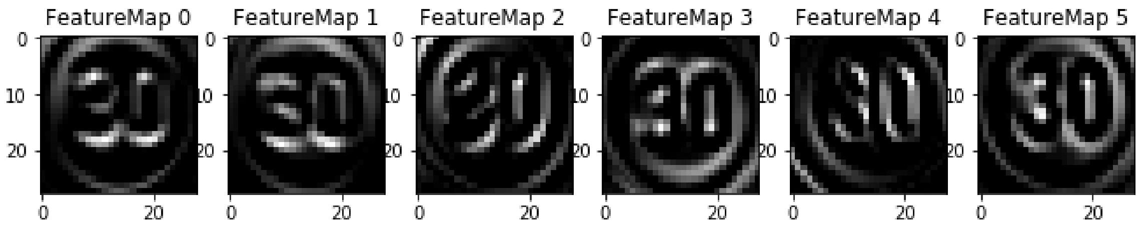
```
INFO:tensorflow:Restoring parameters from ./lenet
```



In [24]:

```
# First convolutional layer relu
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv1_relu = sess.graph.get_tensor_by_name('conv1_relu:0')
    outputFeatureMap(my_signs, conv1_relu)
```

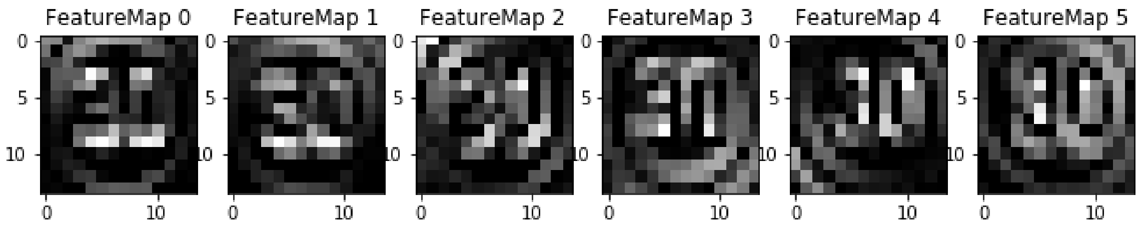
```
INFO:tensorflow:Restoring parameters from ./lenet
```



In [25]:

```
# First convolutional layer pooling
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv1_pool = sess.graph.get_tensor_by_name('conv1_pool:0')
    outputFeatureMap(my_signs, conv1_pool)
```

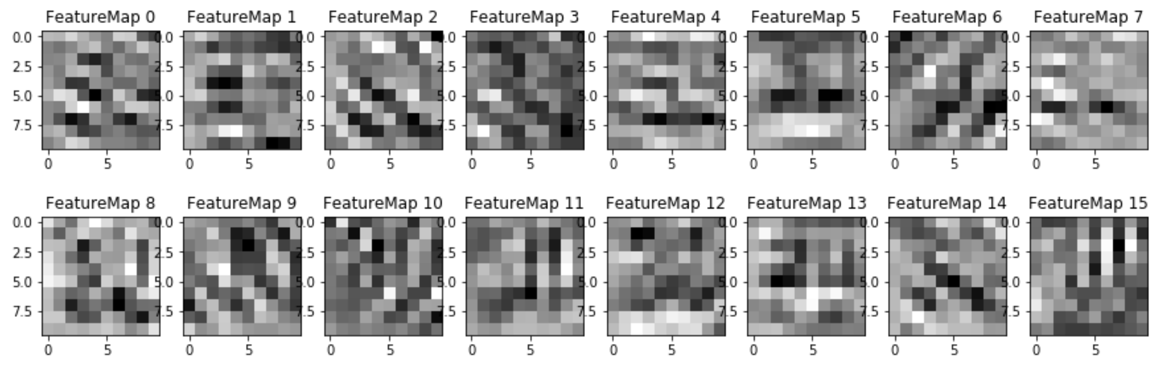
```
INFO:tensorflow:Restoring parameters from ./lenet
```



In [26]:

```
# Second Convolutional layer
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv2 = sess.graph.get_tensor_by_name('conv2:0')
    outputFeatureMap(my_signs, conv2)
```

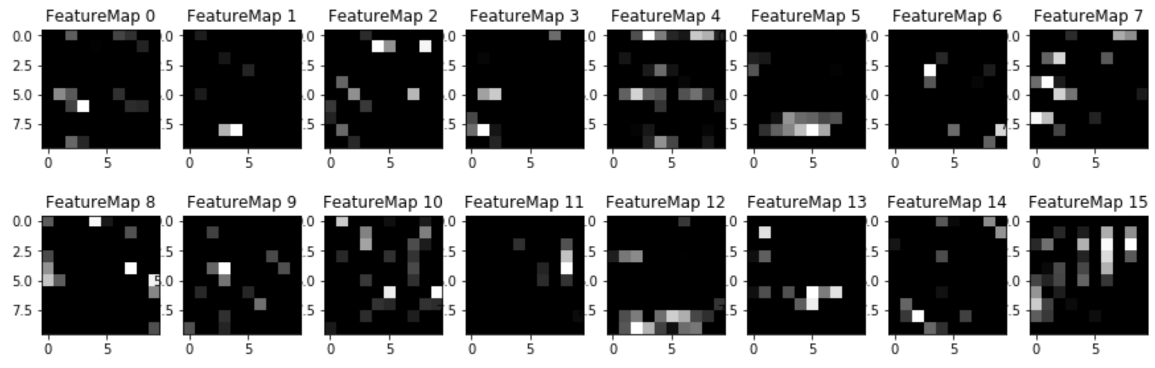
```
INFO:tensorflow:Restoring parameters from ./lenet
```



In [27]:

```
# Second convolutional layer relu
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv2_relu = sess.graph.get_tensor_by_name('conv2_relu:0')
    outputFeatureMap(my_signs, conv2_relu)
```

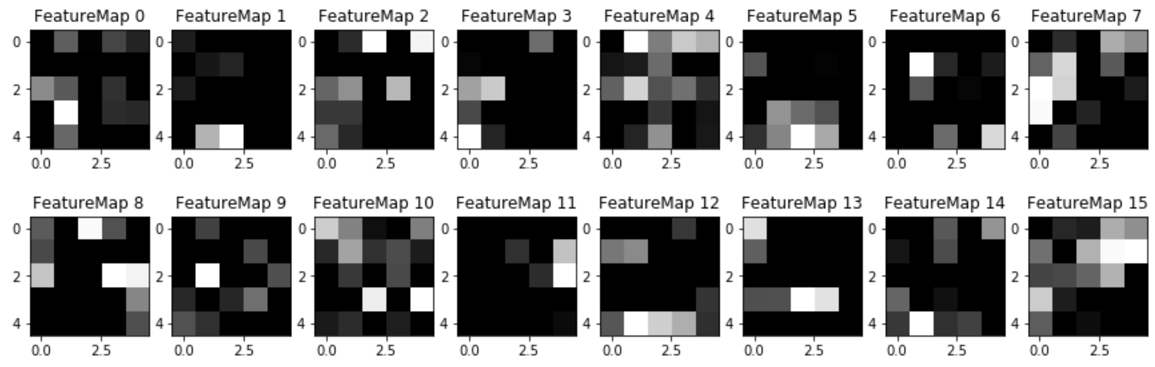
```
INFO:tensorflow:Restoring parameters from ./lenet
```



In [28]:

```
# Second convolutional layer pooling
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    conv2_pool = sess.graph.get_tensor_by_name('conv2_pool:0')
    outputFeatureMap(my_signs, conv2_pool)
```

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


FeatureMap 0
FeatureMap 1
FeatureMap 2
FeatureMap 3
FeatureMap 4
FeatureMap 5
FeatureMap 6
FeatureMap 7
FeatureMap 8
FeatureMap 9
FeatureMap 10
FeatureMap 11
FeatureMap 12
FeatureMap 13
FeatureMap 14
FeatureMap 15