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 [2]:

```
# CELL #1
# Load pickled data
import pickle
import time

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

# Hyper parameters
rate = 0.001
EPOCHS = 30
BATCH_SIZE = 128
kernel_size = 3
dropout = 0.5
newImages = 5

statusFile = 'stat_' + time.strftime("%d_%m_%Y_%H_%M_%S") + '.csv'
log = open(statusFile,'w')

training_file = './data/train.p'
validation_file = './data/valid.p'
testing_file = './data/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]:

```
# CELL #2
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
# TODO: Number of training examples
n_train = len(X_train)

# TODO: Number of testing examples.
n_test = len(X_test)

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

# TODO: How many unique classes/labels there are in the dataset.
n_classes = max(y_test) + 1

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

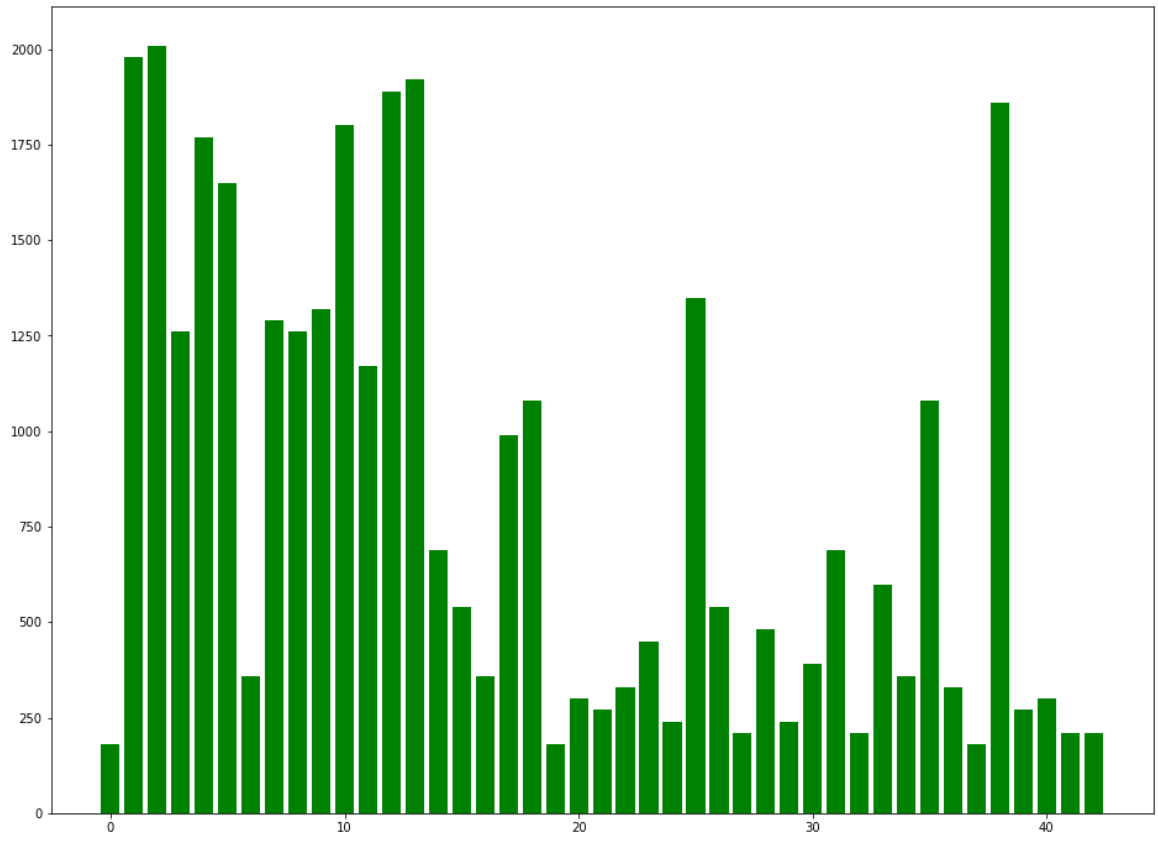
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

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.

In [4]:

```
# CELL #3
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import numpy as np
import random
# Visualizations will be shown in the notebook.
%matplotlib inline
plt.rcParams['figure.figsize'] = (16, 12)
unique, counts = np.unique(y_train, return_counts=True)
fig, ax = plt.subplots()
# Plot visualization for training data distribution
ax.bar(unique, counts, color='g')

#Plot one random image sample of each class

plt.figure()
for i in range(0,n_classes):
    idxVector = np.where(y_train==i)[0]
    randIdx = random.randint(0, len(idxVector) - 1)
    img = X_train[idxVector[randIdx]]
    plt.subplot(6, 8, i+1)
    plt.imshow(img)
```



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 [5]:

```
# CELL #4
### Preprocess the data here. Preprocessing steps could include normalization, conv
### Feel free to use as many code cells as needed.
# Auxiliary image transformations
def rotateImage(img, range):
    rows, cols = img.shape
    M = cv2.getRotationMatrix2D((cols / 2, rows / 2), range * np.random.uniform() -
    img = cv2.warpAffine(img, M, (cols, rows))
    return img

def rollImage(img, range):
    rows, cols = img.shape
    x = range * np.random.uniform() - range / 2
    y = range * np.random.uniform() - range / 2
    M = np.float32([[1, 0, x], [0, 1, y]])
    img = cv2.warpAffine(img, M, (cols, rows))
    return img

def shearImage(img, range):
    rows, cols = img.shape
    pts1 = np.float32([[5, 5], [20, 5], [5, 20]])
    pt1 = 5 + range * np.random.uniform() - range / 2
    pt2 = 20 + range * np.random.uniform() - range / 2
    pts2 = np.float32([[pt1, 5], [pt2, pt1], [5, pt2]])
    M = cv2.getAffineTransform(pts1, pts2)
    img = cv2.warpAffine(img, M, (cols, rows))
    return img

def normalizeGrayscale(img):
    a = -0.5
    b = 0.5
    grayscale_min = 0
    grayscale_max = 255
    return a + (((img - grayscale_min)*(b - a))/(grayscale_max - grayscale_min))
```

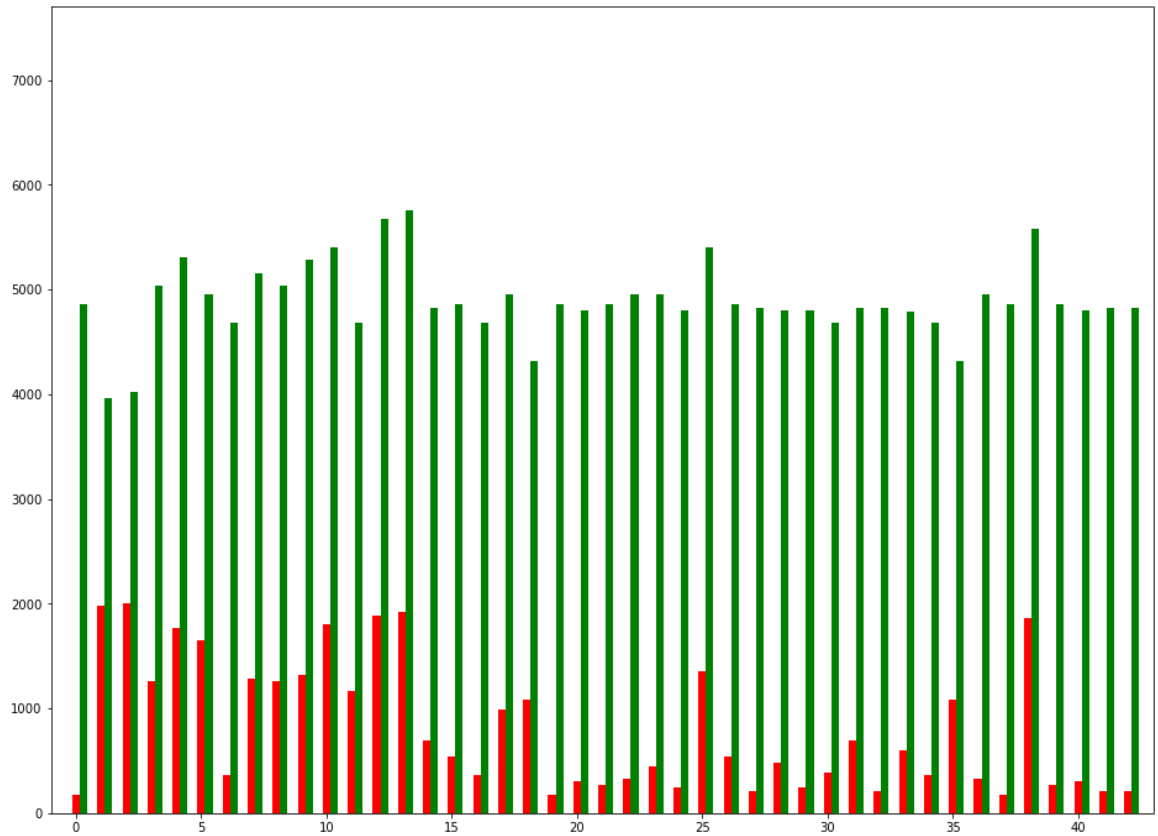
In [6]:

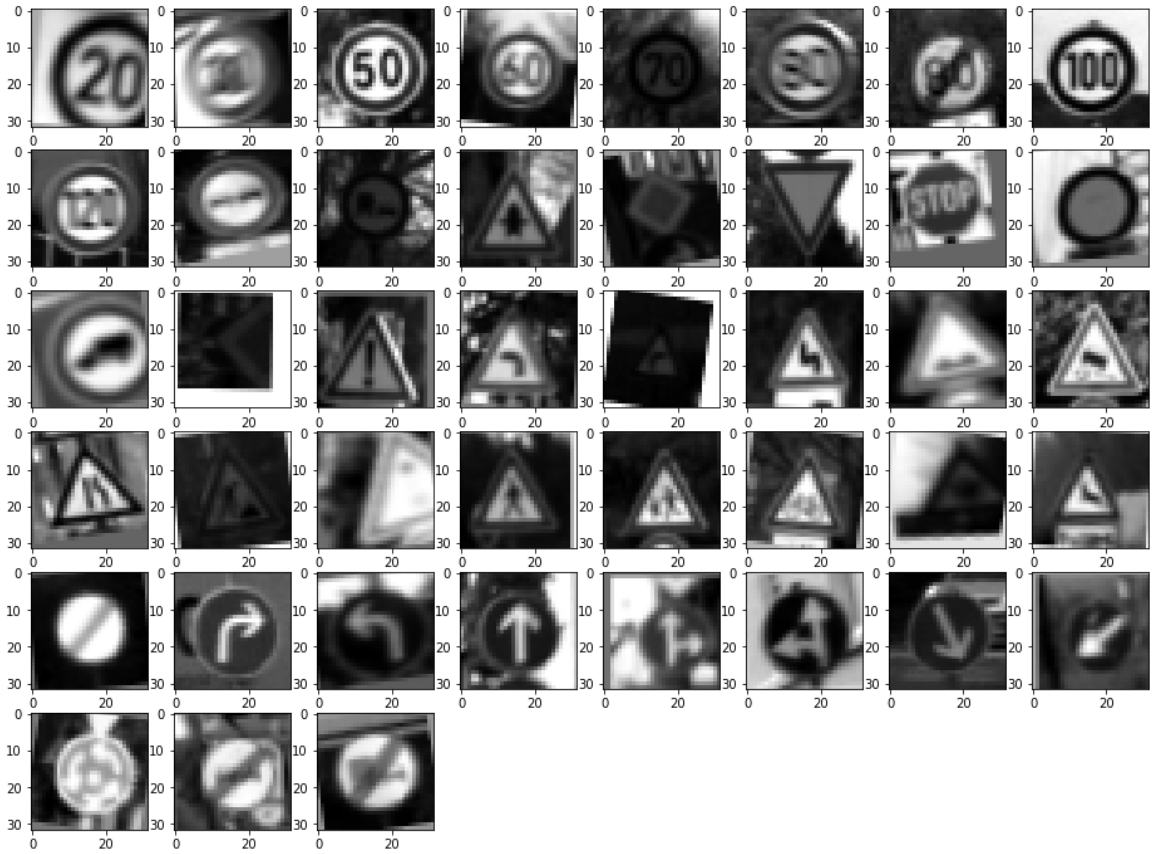
```
# CELL #5
''' Data preprocessing pipeline
The pipeline analises the amount of samples that each class has and based on this i
new images. This stage also convert the images from RGB to gray scale applies norma
The image generator, generates one image with each of the following transformations
1. Image rotate with a random angle between -10 and 10 deg
1. Image rolled a random number of pixels -2.5 and 2.5 pixels
1. Image sheared in the range of -5 to 5
1. Image with gaussian blur
'''
def trainingDataAugmentation(imgVector, yVector):
    unique, counts = np.unique(yVector, return_counts=True)
    targetImgCount = sum(counts)/len(counts) #max(counts)
    expansionMap = []
    generatedImgVector = []
    generatedLabelVector = []
    for label,count in zip(unique, counts):
        expansion = round(((targetImgCount - count) + targetImgCount*newImages)/cou
        if(expansion <= 0.0):
            expansion = newImages
        expansionMap.append(expansion)

    for img,label in zip(imgVector, yVector):
        img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        img = normalizeGrayscale(img)
        generatedImgVector.append(img)
        generatedLabelVector.append(label)
        imgCod = 0
        genCounter = expansionMap[label]
        while(genCounter > 0):
            if(imgCod == 3):
                imgGen = rollImage(img, 5)
                imgGen = shearImage(imgGen, 10)
                imgGen = rotateImage(imgGen, 20)
                imgCod = imgCod + 1
            elif(imgCod == 2):
                imgGen = rollImage(img, 5)
                imgCod = imgCod + 1
            elif(imgCod == 1):
                imgGen = shearImage(img, 10)
                imgCod = imgCod + 1
            elif(imgCod == 0):
                imgGen = rotateImage(img, 20)
                imgCod = imgCod + 1

            if(imgCod > 3):
                imgCod = 0

            genCounter = genCounter - 1
            generatedImgVector.append(imgGen)
            generatedLabelVector.append(label)
    generatedImgVector = np.array(generatedImgVector)[..., np.newaxis]
    generatedLabelVector = np.array(generatedLabelVector)

    return generatedImgVector, generatedLabelVector

def inputImgPreproc(imgVector):
    resultVector = []
    for img in imgVector:
```

```
        img = cv2.cvtColor(img.astype(np.uint8, copy=False), cv2.COLOR_RGB2GRAY)

        img = normalizeGrayscale(img)
        resultVector.append(img)

    return np.array(resultVector)[..., np.newaxis]
```

In [7]:

```
# CELL #6
# Data preprocessing stage
import cv2
import tensorflow as tf
from tensorflow.contrib.layers import flatten
from sklearn.utils import shuffle

# Assert that the images are RGB and then preprocess the images
X_train, y_train = trainingDataAugmentation(X_train, y_train)
X_test = inputImgPreproc(X_test)
X_valid = inputImgPreproc(X_valid)

uniquePostP, countsPostP = np.unique(y_train, return_counts=True)
fig, ax = plt.subplots()
# Plot visualization for training data distribution
width = 0.3
ax.bar(unique, counts, width=width, color='red', label='N', align='center')
ax.bar(uniquePostP + width, countsPostP, width=width, color='green', label='M', ali
ax.axis([-1, 43, 0, 7700])
plt.show()

#Plot one random image sample of each class
plt.figure()
for i in range(0,n_classes):
    idxVector = np.where(y_train==i)[0]
    randIdx = random.randint(0, len(idxVector) - 1)
    img = X_train[idxVector[randIdx]]

    plt.subplot(6, 8, i+1)
    plt.imshow(img.reshape(img.shape[0:2]), cmap='gray')
```





Model Architecture

In [8]:

```
# CELL #7
def evaluateModel(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0X_valid
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+B
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y,
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [9]:

```
# CELL #8
# TensorFlow model auxiliary functions

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')

def maxPool(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VAL

def weightVariable(shape, mu = 0, sigma = 0.1):
    initial = tf.truncated_normal(shape, mean=mu, stddev=sigma)
    return tf.Variable(initial)

def biasVariable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

In [10]:

```
# CELL #9
### Define your architecture here.
'''The network architecture is basically the same as the one presented to solve the
The customization applied is as follow:
1. The traffic sign data set has 43 distinct labels and therefore the output has be
based on 43 classes;
2. After much work with the preprocessing stage, it has been verified that the mode
the data (~74% accuracy on the validation set) and to work around this, dropout has
model in two ways, between each layer and only before the output layer. Both approa
model prediction accuracy and dropout between layers has been selected to the final
3. Even after the addition of dropout, the data was still being overfitted (~82% ac
and this behavior has suggested that a larger network architecture could further im
Thus the first convolutional layer was increased to 32 neurons and the second convo
increased to 64 neurons.
'''
def cnnTSClassifier(x, dropout):
    # First convolutional layer
    # input = 32x32x3
    # output = 14x14x6
    conv1_W = weightVariable([5, 5, 1, 64])
    conv1_b = biasVariable([64])
    conv1 = conv2d(x, conv1_W) + conv1_b
    conv1Relu = tf.nn.relu(conv1)
    conv1MaxPool = maxPool(conv1Relu)
    conv1MaxPool = tf.nn.dropout(conv1MaxPool, dropout)

    # Second convolutional layer
    # input = 14x14x6
    # output = 5x5x16
    conv2_W = weightVariable([5, 5, 64, 128])
    conv2_b = biasVariable([128])
    conv2 = conv2d(conv1MaxPool, conv2_W) + conv2_b
    conv2Relu = tf.nn.relu(conv2)
    conv2MaxPool = maxPool(conv2Relu)
    conv2MaxPool = tf.nn.dropout(conv2MaxPool, dropout)

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0 = flatten(conv2MaxPool)

    # Densely connected layer 1
    fc1_W = weightVariable([5 * 5 * 128, 120])
    fc1_b = biasVariable([120])
    fc1Relu = tf.nn.relu(tf.matmul(fc0, fc1_W) + fc1_b)
    fc1Relu = tf.nn.dropout(fc1Relu, dropout)


    # Densely connected layer 1
    fc2_W = weightVariable([120, 84])
    fc2_b = biasVariable([84])
    fc2Relu = tf.nn.relu(tf.matmul(fc1Relu, fc2_W) + fc2_b)
    fc2Dropout = tf.nn.dropout(fc2Relu, dropout)

    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 43.
    fc3_W = weightVariable([84, 43])
    fc3_b = biasVariable([43])

    logits = tf.matmul(fc2Dropout, 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.

In [11]:

```
# CELL #10
### 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 placeholders allocation
x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)
one_hot_y = tf.one_hot(y, n_classes)

logits = cnnTSClassifier(x, keep_prob)
detailedEvaluation = tf.argmax(logits, 1)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=one_h
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = 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()
```

In [13]:

```
# CELL #11
# Train the model
# Observation: the jupyter notebook is not working with the gpu version of tensorfl
# this cell without the GPU support takes really long but works as well. I'm provid
# as well as a .py file which has the same content as this notebook and runs fine w
statPoints = []
with tf.Session() as sess:
    wallTime = time.time()
    procTime = time.process_time()
    sess.run(tf.global_variables_initializer())

    print("Training...")
    print()
    log.write('Epoch\ttrainingError\tvalidationError\n')
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, n_train, 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_pr

        training_accuracy = evaluateModel(X_train, y_train)
        validation_accuracy = evaluateModel(X_valid, y_valid)

        print("EPOCH {} ...".format(i + 1))
        print("Training Accuracy = {:.3f}".format(training_accuracy))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()
        log.write(str(i)+'\t'+str(1.0 - training_accuracy)+'\t'+str(1.0 - validatio


    saver.save(sess, 'trafficClassifier_normImgplusYUV')
    print("Model saved")
    print('Wall time: {:.4}'.format(time.time() - wallTime))
    print('Processor time: {:.4}'.format(time.process_time() - procTime))
    log.write('#Wall time: {:.4}\n'.format(time.time() - wallTime))
```

```
Training...

EPOCH 1 ...
Training Accuracy = 0.027
Validation Accuracy = 0.048


---------------------------------------------------------------------
------
KeyboardInterrupt                         Traceback (most recent call
 last)
<ipython-input-13-c914a2c91d01> in <module>()
     17             sess.run(training_operation, feed_dict={x: batch_
x, y: batch_y, keep_prob: dropout})
     18
---> 19         training_accuracy = evaluateModel(X_train, y_train)
     20         validation_accuracy = evaluateModel(X_valid, y_valid)
     21

<ipython-input-8-1e40a3ea0f81> in evaluateModel(X_data, y_data)
      6     for offset in range(0, num_examples, BATCH_SIZE):
      7         batch_x, batch_y = X_data[offset:offset+BATCH_SIZE],
```

```
y_data[offset:offset+BATCH_SIZE]
----> 8         accuracy = sess.run(accuracy_operation, feed_dict={x:
 batch_x, y: batch_y, keep_prob: 1.0})
      9         total_accuracy += (accuracy * len(batch_x))
     10     return total_accuracy / num_examples

/home/marcio/.local/lib/python3.5/site-packages/tensorflow/python/cli
ent/session.py in run(self, fetches, feed_dict, options, run_metadat
a)
    765     try:
    766       result = self._run(None, fetches, feed_dict, options_pt
r,
--> 767                          run_metadata_ptr)
    768       if run_metadata:
    769         proto_data =
tf_session.TF_GetBuffer(run_metadata_ptr)

/home/marcio/.local/lib/python3.5/site-packages/tensorflow/python/cli
ent/session.py in _run(self, handle, fetches, feed_dict, options, run
_metadata)
    963     if final_fetches or final_targets:
    964       results = self._do_run(handle, final_targets, final_fet
ches,
--> 965                              feed_dict_string, options, run_m
etadata)
    966     else:
    967       results = []

/home/marcio/.local/lib/python3.5/site-packages/tensorflow/python/cli
ent/session.py in _do_run(self, handle, target_list, fetch_list, feed
_dict, options, run_metadata)
   1013     if handle is None:
   1014       return self._do_call(_run_fn, self._session, feed_dict,
 fetch_list,
-> 1015                            target_list, options, run_metadat
a)
   1016     else:
   1017       return self._do_call(_prun_fn, self._session, handle, f
eed_dict,

/home/marcio/.local/lib/python3.5/site-packages/tensorflow/python/cli
ent/session.py in _do_call(self, fn, *args)
   1020   def _do_call(self, fn, *args):
   1021     try:
-> 1022       return fn(*args)
   1023     except errors.OpError as e:
   1024       message = compat.as_text(e.message)

/home/marcio/.local/lib/python3.5/site-packages/tensorflow/python/cli
ent/session.py in _run_fn(session, feed_dict, fetch_list, target_lis
t, options, run_metadata)
   1002         return tf_session.TF_Run(session, options,
   1003                                  feed_dict, fetch_list, targe
t_list,
-> 1004                                  status, run_metadata)
   1005
   1006     def _prun_fn(session, handle, feed_dict, fetch_list):

KeyboardInterrupt:
```

In [14]:

```
# CELL #12
# Evaluate the model with the test set
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    test_accuracy = evaluateModel(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
    log.write("#Test Accuracy = {:.3f}".format(test_accuracy))
log.close()
```

```
Test Accuracy = 0.969
```

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

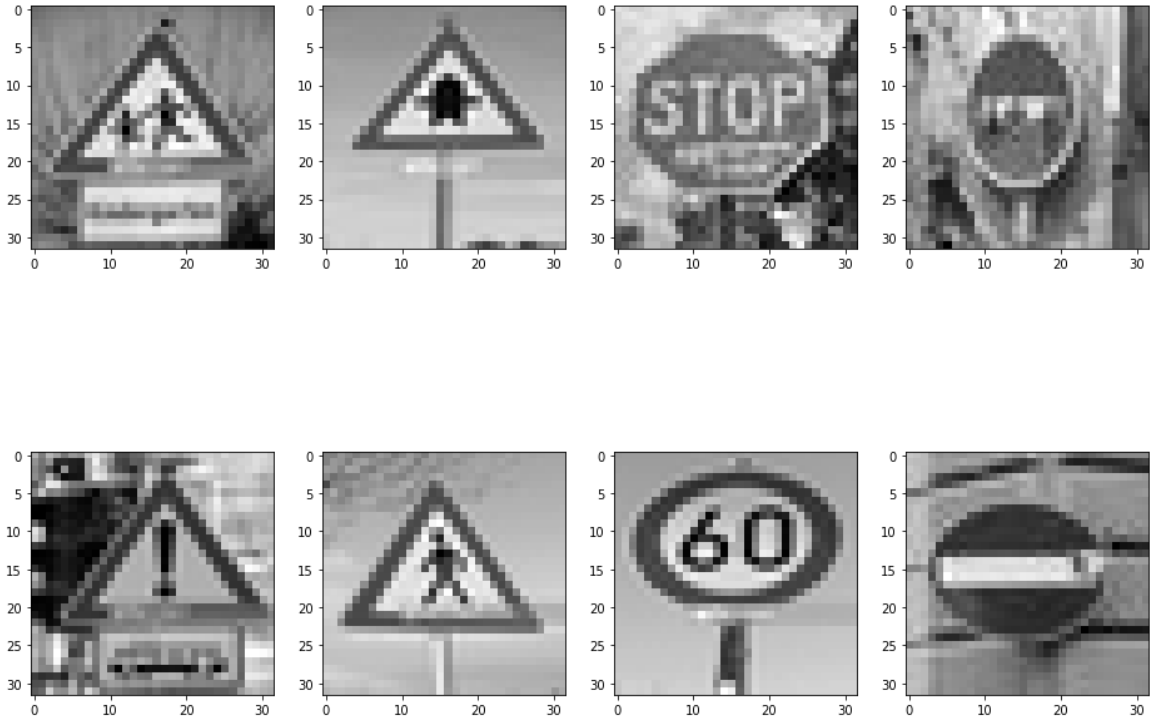
In [15]:

```
# CELL #13
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import os
from scipy import misc
newImagesPath = '/home/marcio/Desktop/projects/carnd/CarND-Traffic-Sign-Classifier-
dirContent = os.listdir(newImagesPath)
imgVec = []
idLabel = []

counter = 0
for imgName in dirContent:
    img = misc.imread(newImagesPath+imgName)
    imgVec.append(img)
    idLabel.append(int(imgName.split('_')[0]))
    plt.subplot(2, 4, counter+1)
    counter = counter + 1
    plt.imshow(img)
```

Predict the Sign Type for Each Image

In [16]:

```
# CELL #14
### Run the predictions here and use the model to output the prediction for each im
### Make sure to pre-process the images with the same pre-processing pipeline used
### Feel free to use as many code cells as needed.
imgVec = inputImgPreproc(imgVec)
counter = 0
for img in imgVec:
    plt.subplot(2, 4, counter+1)
    counter = counter + 1
    plt.imshow(img.reshape(img.shape[0:2]), cmap='gray')

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    prediction = sess.run(detailedEvaluation, feed_dict={x: imgVec, keep_prob: 1.0}
    probVec = sess.run(tf.nn.top_k(sess.run(logits, feed_dict={x: imgVec, keep_prob
```



Analyze Performance

In [17]:

```
# CELL #15
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accura
correct = 0
for pred,ans in zip(prediction, idLabel):
    if pred == ans:
        correct = correct + 1

print("Model accuracy with new images = {:.3f}%".format(100*correct/len(prediction)
```

```
Model accuracy with new images = 87.500%
```

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,  0.07893
497,
         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 [18]:

```
# CELL #16
### Print out the top five softmax probabilities for the predictions on the German
### Feel free to use as many code cells as needed.
# Already implemented on the previous cell
print(probVec)
```

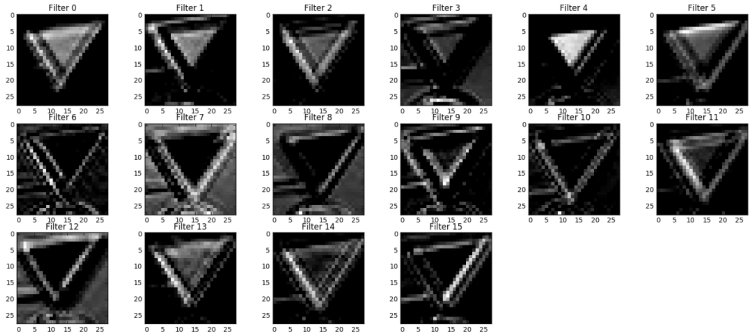
```
TopKV2(values=array([[ 17.00011635,   3.59490514,   3.47018385,   2.7
2490335,
          2.29449701],
       [ 20.50906754,   7.16284847,   4.99779272,   1.70046639,
          1.11199272],
       [ 18.21655655,  -1.66229117,  -2.26371646,  -4.12604952,
         -4.21079922],
       [  8.04035282,   6.35569811,   4.40921688,   3.11303568,
          1.59483194],
       [ 25.4187603 ,   8.05271149,   5.97021961,   5.75660181,
          5.36177588],
       [ 32.38572693,  19.5124855 ,  19.48689461,   6.6344676 ,
          5.87995529],
       [  5.1326375 ,   2.48794627,   2.03955412,   0.79107308,
          0.58216363],
       [ 31.48065567,   7.97179222,   5.1098156 ,   5.08313417,
          4.64619541]], dtype=float32), indices=array([[28, 11, 20, 2
3, 30],
       [11, 30, 27, 36, 28],
       [14, 38,  0,  9, 17],
       [14, 17, 40, 12, 38],
       [18, 26, 22, 29, 27],
       [27, 11, 18, 26, 24],
       [ 3, 16,  0,  9,  5],
       [17, 34, 38, 14, 16]], dtype=int32))
```

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 [ ]:

```
# CELL #17
### 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 map
# tf_activation: should be a tf variable name used during your training procedure t
# activation_min/max: can be used to view the activation contrast in more detail, b
# plt_num: used to plot out multiple different weight feature map sets on the same

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_max=
    # 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
    # If you get an error tf_activation is not defined it maybe having trouble acce
    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
        plt.title('FeatureMap ' + str(featuremap)) # displays the feature map numbe
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap
```

Question 9