Self-Driving Car Engineer Nanodegree

Project: Build a Traffic Sign Recognition Classifier

This project provides a solution to Project 2 of the Udacity Self Driving Car Nanodegree. The goal of the project was to build a NN to classify signs using the German Traffic Sign Dataset (http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

The following steps were taken, and are summarized in the sections below:

- Load the data set
- Explore, summarize and visualize the data set
- Design, train and test a model architecture
- Use the model to make predictions on new images
- Analyze the softmax probabilities of the new images
- Summarize the results with a written report

Instructions for the project can be found here (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/README.md)

The rubric for the project can be found here (https://review.udacity.com/#!/rubrics/481/view)

Step 0: Load The Data

Data was provided by Udacity in pickled format.

In [1]:
```
# Load pickled data
import pickle
import numpy as np
import cv2

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_raw, y_train_raw = np.array(train['features'], np.uint8), np.array(tra
in['labels'])
X_valid_raw, y_valid_raw = np.array(valid['features'], np.uint8), np.array(val
id['labels'])
X_test_raw, y_test_raw = np.array(test['features'], np.uint8), np.array(test[
'labels'])
```

Step 1: Dataset Summary & Exploration

The data that was just loaded 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**

In [2]:
```
# Number of training examples
n_train = len(y_train_raw)

# Number of validation examples
n_valid = len(y_valid_raw)

# Number of testing examples.
n_test = len(y_test_raw)

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

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

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 =", img_shp)
print("Number of classes =", n_class)
```

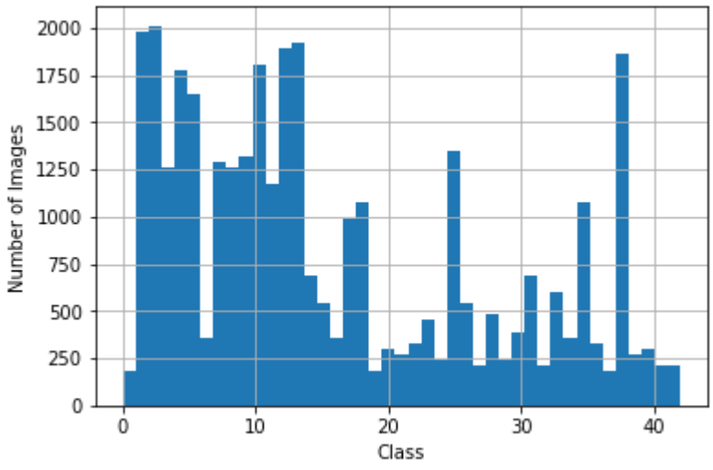
```
Number of training examples = 34799
Number of validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Visual Exploration

Here, some figures will be plotted to provide a visual summary of the dataset.

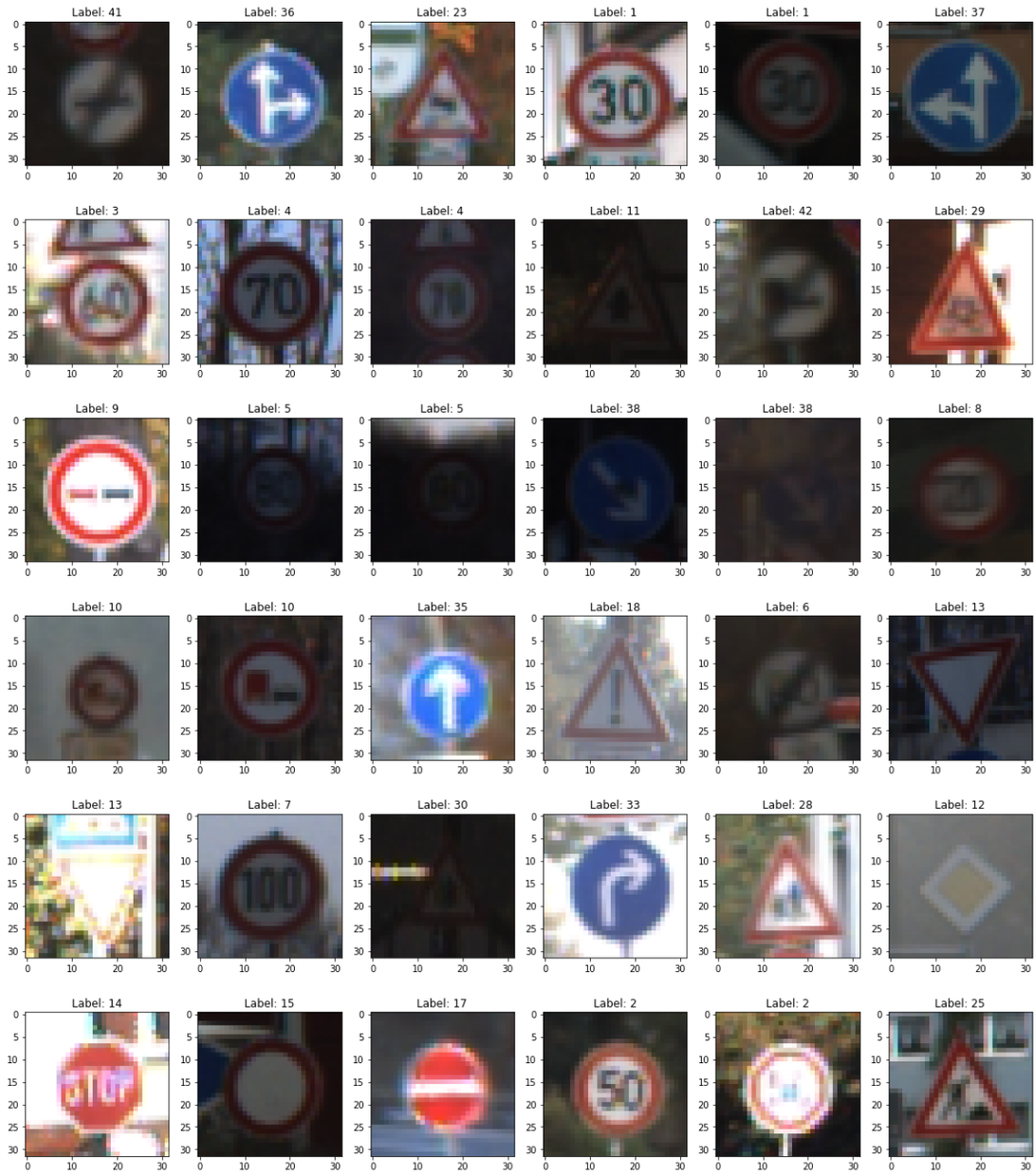
In [3]:
```
# Plotting utilities that will be used throughout this notebook are defined he
re

import math
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline


def hist_plot(y):
    plt.hist(y, n_class);
    plt.xlabel('Class');
    plt.ylabel('Number of Images');
    ax = plt.gca();
    ax.grid(True)
    plt.show();


def plot_imgs(X, y=np.array([]), cols = 6, cmap='brg'):

    num_cols = cols
    num_plots = len(X)
    num_rows = int(math.ceil(num_plots/2))

    plotNum = 1
    plt.figure(figsize = (20, num_rows*4))
    for i in range(num_plots):

        plt.subplot(num_rows, num_cols, plotNum)
        plt.imshow(X[i], cmap=cmap)
        if(y.size > 0):
            plt.title("Label: " + str(y[i]))
        plotNum = plotNum + 1
```

In [4]:
```
# Histogram of data
hist_plot(y_train_raw)
```



As you can see, there is a significant disparity in the number of images across classes. This hints at the need for supplementation of images across classes to narrow down this gap, otherwise, the model may be biased towards the signs with the most samples. Obviously, some signs are more rare than others, but in the case of driving, it wouldn't be acceptable to simply fail on rare signs; the classifier should ideally classify all signs correctly.

In [5]:
```
# Pick and plot some example test images
test_img_labels = np.arange(0, n_train, 975)
test_imgs = X_train_raw[test_img_labels]

# Pick randomly from the dataset
plot_imgs(X_train_raw[test_img_labels], y_train_raw[test_img_labels])
```



The most noticeable aspect of the images here is that they vary greatly in luminosity. This is to be expected, as self-driving cars can't be expected to only operate in the day time. This hints at the need for some sort of luminosity correction in the image processing pipeline.

Step 2: Design and Test a Model Architecture

Dataset Preprocessing

Two factors contributed to the development of the pre-processing pipeline below:

1. The large disparity in the number of samples across classes, which has the potential to bias the NN during training, and
2. The vast differences in luminosity across samples.

To deal with these issues, I developed a method to supplement images by randomly warping them, and then performed a luminosity correction.

The warping pipeline consists of a randowm rotation and a random affine warping.

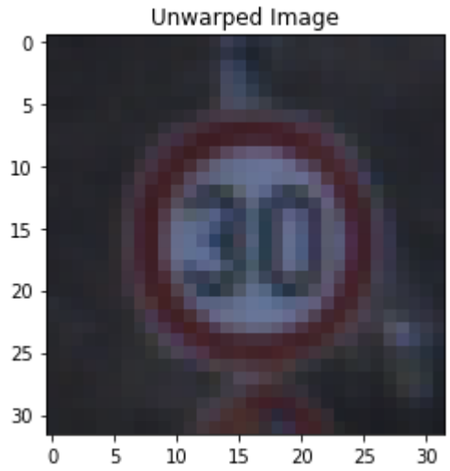
The luminosity correctoin uses an adaptive localized Clahe histogram equalization method (https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html) on the Y layer of each image (after conversion to YUV).

NOTE: I've saved off data that's been processed. If you're loading up this project, you will need to re-generate the processed images, by setting the src variable below to "gen."

In [6]:
```
# Img warping pipeline

def warp_img(X):

    rotate = 12
    warp = 1.8

    size = img_shp[0]

    # Rotation
    M = cv2.getRotationMatrix2D((size/2,size/2), (np.random.random(1)-0.5)*rot
ate, 1)
    X = cv2.warpAffine(X, M, dsize=(size,size), borderMode= cv2.BORDER_REPLICA
TE)


    # Affine Warping
    p_src = np.float32([[warp,warp], [warp, 32-warp], [32-warp, warp]])
    p_dst = np.copy(p_src)

    for i in range(len(p_dst)):
        p_dst[i] = p_dst[i] + (np.random.random(2)-0.5)*2.0*warp

    M = cv2.getAffineTransform(p_src, p_dst)
    X = cv2.warpAffine(X, M, dsize=(size,size), borderMode= cv2.BORDER_REPLICA
TE)

    # Blurring
    blur = np.random.choice([1,3])
    X = cv2.GaussianBlur(X, ksize=(blur,blur), sigmaX = 0.4, sigmaY = 0.4)

    return X
```

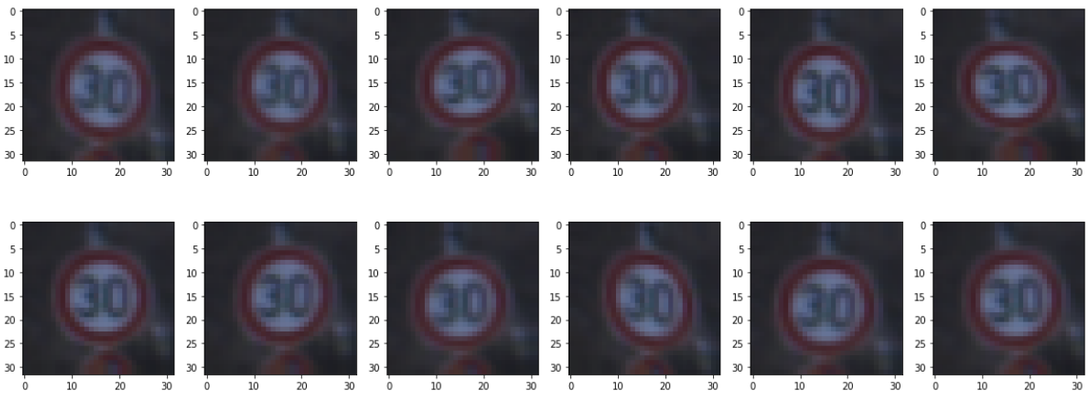
Below is an example of a an unwarped image.

In [7]:
```
# Quick test and visualization of warping agorithm
test_img = X_train_raw[4000]

plt.figure()
plt.title('Unwarped Image')
plt.imshow(test_img);
```



Below are several examples of that same image run through the warping pipeline.

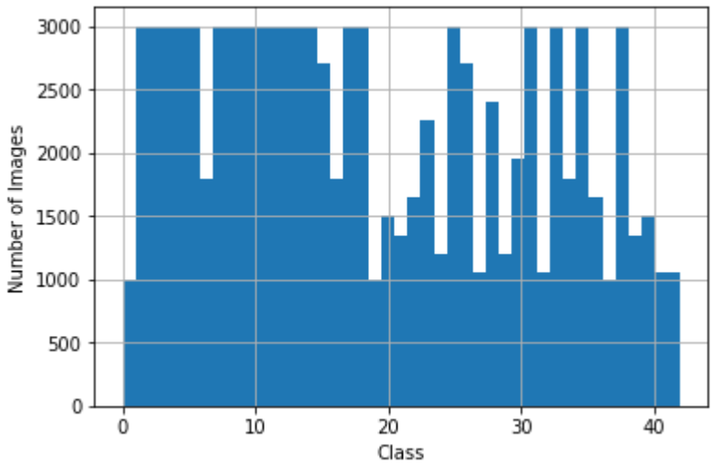
In [8]:
```
# Sample of warped images

warped_imgs= np.empty([12,test_img.shape[0], test_img.shape[0], 3], dtype=test
_img.dtype)
for i in range(12):
    warped_imgs[i] = warp_img(test_img)


plot_imgs(warped_imgs);
```



In [9]:
```
# Augment data set

def suppl_img(X, y):

    hist, edges = np.histogram(y, bins=n_class)

    X_new = np.empty([0, X.shape[1], X.shape[2], X.shape[3]], dtype=X.dtype)
    y_new = np.empty([0], dtype = y.dtype)

    for i in range(n_class):

        num_imgs = hist[i]

        # Minimum of 1000 images, max of 3000.
        min_imgs = np.maximum(num_imgs*5, 1000)
        min_imgs = np.minimum(min_imgs, 3000)
        print("Number of images in class ", i, ": ", num_imgs)

        if num_imgs < min_imgs:
            label = i
            imgs = X[y == label]

            num_new = min_imgs - num_imgs
            print("Number of new images for class", i, ":", num_new)

            for j in range(num_new):
                # Pick a random image from the class
                rand_num = np.random.randint(0,num_imgs)
                rand_img = imgs[np.random.randint(0,num_imgs)]

                # Apply warping function
                rand_img = warp_img(rand_img)

                # Reshape for append
                rand_img = np.reshape(rand_img, (1,32,32,3))

                # Append to new images
                X_new = np.append(X_new, rand_img, axis = 0)
                y_new = np.append(y_new, [label], axis = 0)

    ## Final image set
    X = np.append(X, X_new, axis = 0)
    y = np.append(y, y_new, axis = 0)
    return (X, y)



# Choose to generate images, load images from file, or skip image generation
 (i.e.: use baseline image set)
src = "load"

X_save = 'X_train_suppl.p'
y_save = 'y_train_suppl.p'

if(src == "gen"):
    X_train_aug, y_train_aug = suppl_img(X_train_raw, y_train_raw)
    pickle.dump(X_train_aug, open( X_save, "wb" ))
    pickle.dump(y_train_aug, open( y_save, "wb" ))
elif (src == "load"):
    with open(X_save, mode='rb') as f:
        X_train_aug = pickle.load(f)
    with open(y_save, mode='rb') as f:
        y_train_aug = pickle.load(f)
elif (src == "skip"):
    X_train_aug = X_train_raw
    y_train_aug = y_train_raw
else:
    print("Where's your data, bro?")
```

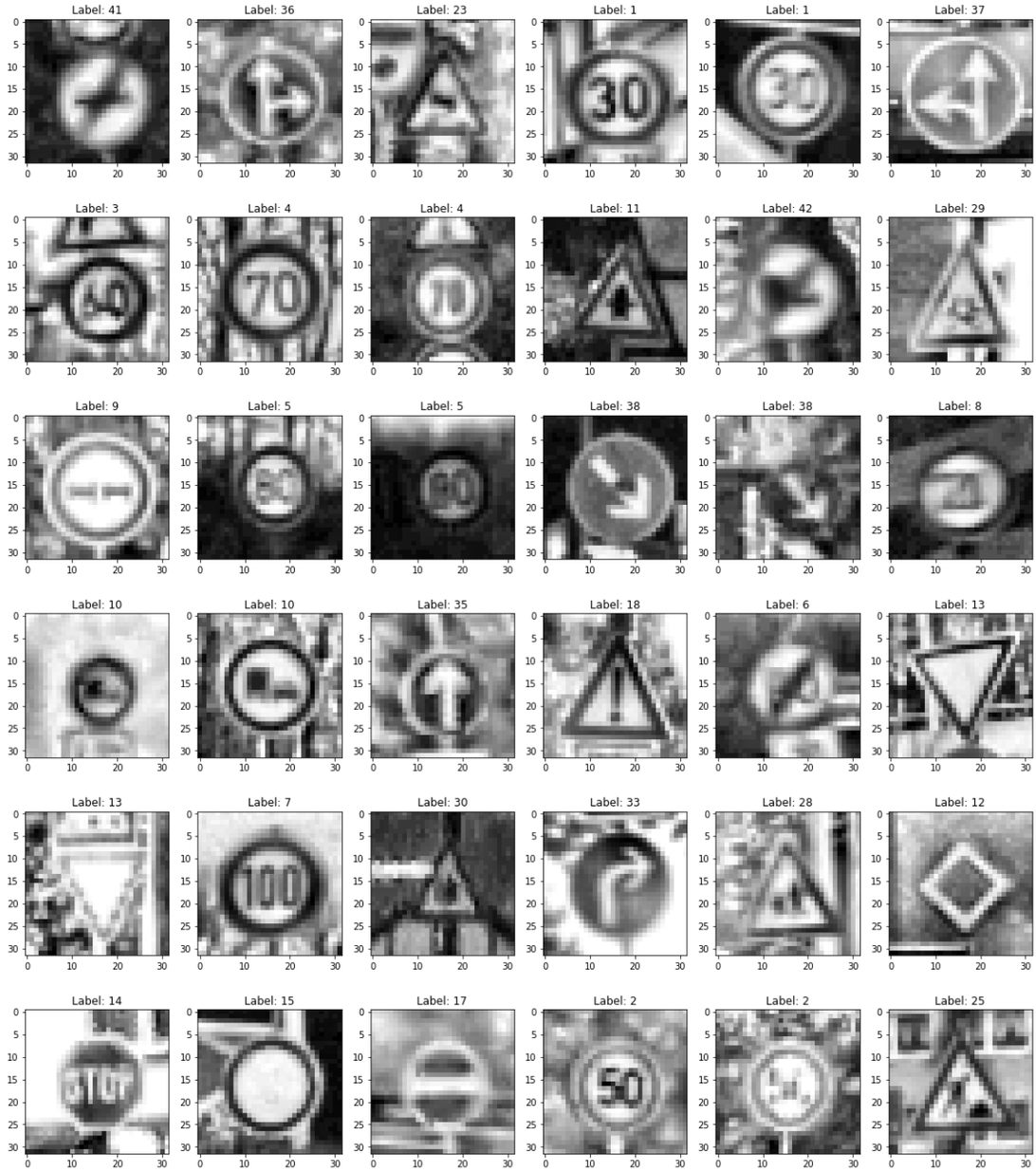
Below is a histogram of the images after supplemenation. As you can see, the disparity between images has been narrowed.

In [10]:
```
hist_plot(y_train_aug)
```



The section below performs a luminosity correction and converts the images to grayscale.

In [11]:
```
def lumin_correct(X):
    s = X.shape
    X_new = np.ndarray((s[0], s[1], s[2], s[3]), np.uint8)
    clahe = cv2.createCLAHE(clipLimit=2, tileGridSize=(4,4))
    for i in range(len(X)):
        X_new[i] = cv2.cvtColor(X[i], cv2.COLOR_RGB2YUV)
        X_new[i,:,:,0] = clahe.apply(X_new[i,:,:,0])
        #X_new[i,:,:,0] = cv2.equalizeHist(X_new[i,:,:,0])
        X_new[i] = cv2.cvtColor(X_new[i], cv2.COLOR_YUV2RGB) # could go straig
ht to grayscale from here...
    return X_new


def grayscale(X):
    s = X.shape
    X_new = np.ndarray((s[0], s[1], s[2]), np.uint8)
    for i in range(len(X)):
        X_new[i] = cv2.cvtColor(X[i], cv2.COLOR_RGB2GRAY)

    return X_new

def rshp(X):
    return X.reshape(X.shape + (1,))


def process_img(X):

    X = lumin_correct(X)
    X = grayscale(X)

    return X

X_train_proc = process_img(X_train_aug)
X_valid_proc = process_img(X_valid_raw)
X_test_proc = process_img(X_test_raw)


plot_imgs(X_train_proc[test_img_labels], y_train_raw[test_img_labels], cmap =
'gray')
```



Here, the images are normalized to be within the range -1.0 to 1.0

In [12]:
```
# Normalize and reshape

def normalize(X):
    X = (np.float32(X)-128.0)/128.0
    return X

X_train = normalize(X_train_proc)
X_valid = normalize(X_valid_proc)
X_test = normalize(X_test_proc)
```

In [13]:
```
# Set up training data
# The image processing pipeline convered images to 32x32; need to add another
 dimension for TF.
X_train = rshp(X_train)
X_valid = rshp(X_valid)
X_test = rshp(X_test)

y_train = y_train_aug
y_valid = y_valid_raw
y_test = y_test_raw

# NOTE: I've purposely kept variables in memory throught the process.
# This was done simply for ease of use and plotting (i.e.: older, unprocessed
 variants of the dataset).
# If RAM/memory were an issue, I would have cleared variables after they were
 used.
```

Model Architecture

I based my NN architecture on that developed by Sermanet and LeCun (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf), and referenced in the Udacity coursework, with some modifications to the classification layer. However, I didn't have the ability (i.e.: time or computer power) to run networks of the depth that were proposed by Sermenet and LeCun, and thus, the network below as fewer convolutional layers.

The classifier is based on LeNet with adjustments made to account for increased matrix sizes. Four different rates of dropout were applied to different layers in the architecture; this was based on findings that high rates of dropout in the largest connected layers were beneficial, but those same high rates of dropout in the convolutional and smaller connected layers were detrimental.

In [14]:
```
import tensorflow as tf
from tensorflow.contrib.layers import flatten


# Helper function to create a fully connected layer
def conctdLayer(x, mu, sigma, in_dim, out_dim):
    w = tf.Variable(tf.truncated_normal([in_dim, out_dim], mean = mu, stddev =
 sigma))
    b = tf.Variable(tf.zeros([out_dim]))
    x1 = tf.add(tf.matmul(x, w), b)

    return x1


# Helper function to create a convolution layer
def convLayer(x, mu, sigma, fsize, stride, pad, keep):
    w = tf.Variable(tf.truncated_normal(fsize, mean = mu, stddev = sigma))
    b = tf.Variable(tf.zeros(fsize[3]))

    x1 = tf.nn.conv2d(x, w, strides=[1,stride,stride,1], padding = pad)
    x1 = tf.nn.bias_add(x1, b)

    x1 = tf.nn.relu(x1)
    x1 = tf.nn.dropout(x1, keep)

    return x1


# Covnet
def covnet(x):
    mu = 0
    sigma = 0.1

    # Convolution -> 32x32 to 28x28
    x = convLayer(x, mu, sigma, [5, 5, 1, 10], 1, 'VALID', keep[0])
    print("After conv 1",x.get_shape())


    # Max pool -> 28x28 to 14x14
    x1 = tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], paddi
ng = 'VALID')
    print("After maxpool 1",x1.get_shape())

    # Convolution -> 14x14 to 10x10
    x2 = convLayer(x1, mu, sigma, [5, 5, 10, 20], 1, 'VALID', keep[1])
    print("After conv 2:",x2.get_shape())

    # Max pool -> 10x10 to 5x5
    x2 = tf.nn.max_pool(x2, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padd
ing = 'VALID')
    print("After maxpool 3:",x2.get_shape())

    x3 = convLayer(x2, mu, sigma, [5, 5, 20, 500], 1, 'VALID', keep[1])
    print("After conv 3:",x3.get_shape())

    # Flatten and concat -> 1x900
    x = tf.concat([tf.contrib.layers.flatten(x3), tf.contrib.layers.flatten(x1
)], 1)
    print("After flatten:",x.get_shape())

    # Connected layer -> 900x120
    x = conctdLayer(x, mu, sigma, 2460, 120)
    x = tf.nn.relu(x)
    x = tf.nn.dropout(x, keep[2])

    # Connected layer -> 3216x120
    x = conctdLayer(x, mu, sigma, 120, 84)
    x = tf.nn.relu(x)
    x = tf.nn.dropout(x, keep[3])

    # Connected layer -> 120x43
    x = conctdLayer(x, mu, sigma, 84, 43)
```

```
    return x
```

Train, Validate and Test the Model

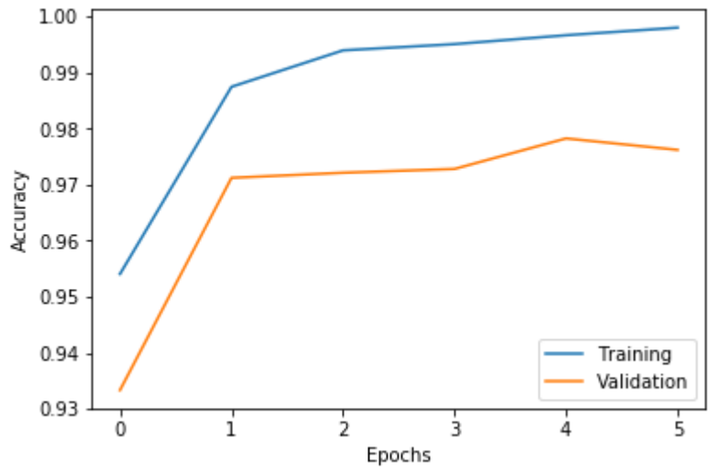
In [15]:
```
from sklearn.utils import shuffle


# Set up parameters for training
EPOCHS = 6
BATCH_SIZE = 128
LEARN_RATE = 0.0008


x = tf.placeholder(tf.float32, (None, img_shp[0], img_shp[1], 1))
y = tf.placeholder(tf.float32, (None))
keep = tf.placeholder(tf.float32, 4)
one_hot_y = tf.one_hot(tf.cast(y, tf.int32), tf.cast(n_class, tf.int32))


logits = covnet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logi
ts=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer()
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()

# Helper function for evaluating accruacy
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:off
set+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batc
h_y, keep: (1.0, 1.0, 1.0, 1.0)})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples

# Variables for plotting a learning curve across epochs
training_accuracy = np.zeros(EPOCHS)
validation_accuracy = np.zeros(EPOCHS)


# Start the session and train
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:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, ke
ep: (0.8, 0.95, 0.5, 1.0)})

        training_accuracy[i] = evaluate(X_train, y_train)
        validation_accuracy[i] = evaluate(X_valid, y_valid)

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

    saver.save(sess, './lenet')

    print("Model saved")
```

```
After conv 1 (?, 28, 28, 10)
After maxpool 1 (?, 14, 14, 10)
After conv 2: (?, 10, 10, 20)
After maxpool 3: (?, 5, 5, 20)
After conv 3: (?, 1, 1, 500)
After flatten: (?, 2460)
WARNING:tensorflow:From <ipython-input-15-c6b0c31176fd>:17: softmax_cross_ent
ropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will b
e 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}.

Training...

EPOCH 1 ...
Training Accuracy = 0.954
Validation Accuracy = 0.933

EPOCH 2 ...
Training Accuracy = 0.987
Validation Accuracy = 0.971

EPOCH 3 ...
Training Accuracy = 0.994
Validation Accuracy = 0.972

EPOCH 4 ...
Training Accuracy = 0.995
Validation Accuracy = 0.973

EPOCH 5 ...
Training Accuracy = 0.997
Validation Accuracy = 0.978

EPOCH 6 ...
Training Accuracy = 0.998
Validation Accuracy = 0.976

Model saved
```

Here, I'm plotting a learning curve to illustrate how well the model is performing on the training vs validation data. Note that I've modified the number of epochs that are run here based on an initial run of 10 epochs; the learning curve for that initial run can be found in the README.md file.

In [16]:
```
# Plot learning curve
plt.figure();

plt.plot(training_accuracy);
plt.plot(validation_accuracy);

plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['Training', 'Validation'])
```

Out[16]: <matplotlib.legend.Legend at 0x3fafb438>



In [17]:
```
# Check test accuracy
with tf.Session() as sess:
##sess = tf.get_default_session()
    sess.run(tf.global_variables_initializer());
    saver2 = tf.train.import_meta_graph('./lenet.meta');
    saver2.restore(sess, "./lenet");

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

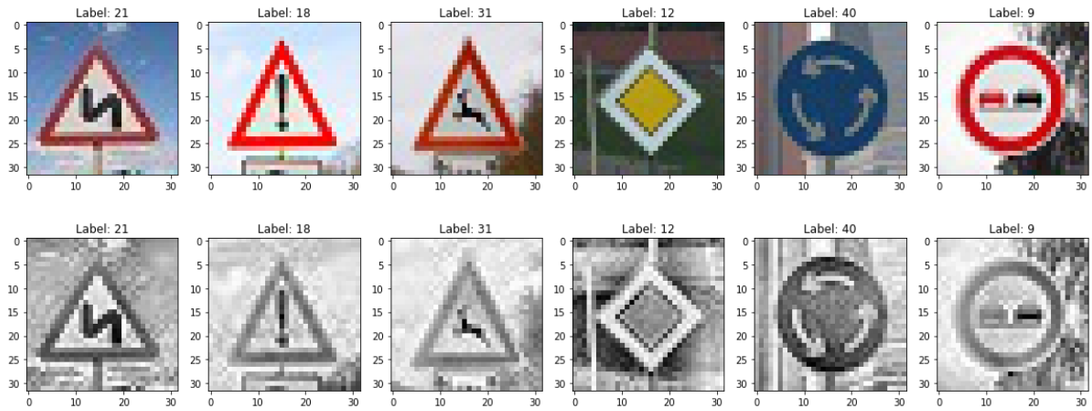
```
INFO:tensorflow:Restoring parameters from ./lenet
Test Accuracy = 0.961
```

Step 3: Test a Model on New Images

In this section, I selected 6 German traffic signs from the web to test how well the network was able to classify them. I chose 3 images of the same shape (triangle with red border) to test how well the network was able to distinguish the elements in the center of the sign. The others were chosen at random. In general, I felt all should be rather simple to classify, as signs were easy to visibly distinguish (unlike some signs in the original training set), centered in the image, had good lighting, and were not at all blurry.

Load and Output the Images

In [18]:
```
import glob

img_files = glob.glob('my_imgs/sign*.jpg')

my_labels = np.array([21, 18, 31, 12, 40, 9])
my_imgs_raw = []
for file in img_files:
    my_imgs_raw.append(plt.imread(file))

my_imgs_raw = np.array(my_imgs_raw)
plot_imgs(my_imgs_raw, my_labels)

my_imgs = process_img(my_imgs_raw)
my_imgs = normalize(my_imgs)

plot_imgs(my_imgs, my_labels, cmap = 'gray')

my_imgs = rshp(my_imgs)
```



Predict the Sign Type for Each Image and Output Top 5 Softmax Probabilities

Below, I've shown the prediction accuracy for the test images, and have plotted a figure that shows the top 5 softmax probabilities (i.e.: top 5 guesses) for each of the test images.

In [19]:
```
my_softmax = tf.nn.softmax(logits)
my_top_k = tf.nn.top_k(my_softmax, k=5)


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer());
    saver3 = tf.train.import_meta_graph('./lenet.meta');
    saver3.restore(sess, "./lenet");

    my_img_prob = sess.run(my_softmax, feed_dict={x: my_imgs, keep: (1.0, 1.0,
 1.0, 1.0)})
    my_top_k = sess.run(my_top_k, feed_dict={x: my_imgs, keep: (1.0, 1.0, 1.0,
 1.0)})

    my_acc = evaluate(my_imgs, my_labels)


    print("Accuracy for my images: {:.3f}".format(my_acc))
```

```
INFO:tensorflow:Restoring parameters from ./lenet
Accuracy for my images: 0.833
```

In [20]:
```
ind = np.array(my_top_k.indices)
prob = np.array(my_top_k.values)

rows = ind.shape[0]
cols = ind.shape[1]

plt.figure(figsize=(20,20))
plotNum = 1;
for i in range(rows):
    plt.subplot(rows, cols+1, plotNum)
    plt.imshow(my_imgs_raw[i])
    plt.title("Test Image")
    plotNum = plotNum + 1
    for j in range(cols):
        img_idx = ind[i,j]
        img_prob = prob[i,j]
        ex_img = X_test_raw[y_test_raw == img_idx][10] ## pick an image from t
he test set...
        plt.subplot(rows, cols+1, plotNum)
        plt.imshow(ex_img)
        plt.title("Label: {:d}; Prob: {:.8f}".format(img_idx, img_prob))
        plotNum = plotNum + 1
```

See the README.md file for an analalysis of all of the images. Below, I will focus on the incorrectly classified "double curve" image.

Analysis of Incorrect "Double Curve" Classification

Surprisingly, the classifier was unable to classify the double curve sign correctly. The originial image I selected from the web was actually flipped 180 degrees at first (i.e.: double curve to the right). Looking at the training set (shown below), it appears as if the only included samples have a "double curve to the left". I assumed that flipping the image by 180 degrees would then result in a correct classification; however, the classifier still failed, classifying the sign as "children crossing". It appears thie is because the sign I selected is a slight variant of the sign in the training set. Note that the signs in the training set have a shorter middle siection, such that the two vertical sections of the "double curve" are closer together. This is an interesting in that it demonstrates potential issues with NNs, and the importance of having a varied training set.

NOTE: it's also apparent that the training set contained a rather small number of variations of the sign in quetion (i.e.: the samples were derived from a small number of "passes" of the "double curve" sign; I assume that a more varied training set may have improved the outcome.

In [21]:
```
imgs21 = X_train_raw[y_train_raw == 21][np.arange(0,270,8)]
plot_imgs(imgs21)
```