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. Sections that begin with **'Implementation'** in the header indicate where you should begin your implementation for your project. Note that some sections of implementation are optional, and will be marked with **'Optional'** in the header.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **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 [38]:
```
# Load pickled data
import pickle

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

training_file = "data/train.p"
testing_file = "data/test.p"

with open(training_file, mode='rb') as f:
    train = pickle.load(f)
with open(testing_file, mode='rb') as f:
    test = pickle.load(f)

X_train, y_train = train['features'], train['labels']
X_test, y_test = test['features'], test['labels']
set(y_train)
X_test.shape
```

Out[38]:
```
(12630, 32, 32, 3)
```

In [2]:
```
# Load all the packages
import numpy as np
import csv
import matplotlib.pyplot as plt
import time
import tensorflow as tf
import cv2
import random
from sklearn.utils import shuffle, resample
```

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 2D 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.

In [3]:
```
with open('signnames.csv', 'r') as csvfile:
    signnames = csv.DictReader(csvfile)
    SignNames = []
    for row in signnames:
        SignNames.append(row['SignName'])
        #print(row['ClassId']+' : ' + row['SignName'])
    #print(SignNames[0])
```

In [4]:

```
### Replace each question mark with the appropriate value.
# TODO: Number of training examples
# Done
n_train = X_train.shape[0]

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

# TODO: What's the shape of an traffic sign image?
# Done
image_shape = X_train.shape[1:4]

# TODO: How many unique classes/labels there are in the dataset.
# Done
n_classes = np.unique(y_train).size

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)

for i in range(n_classes):
    print("Class {} ({}) has {} training samples and {} testing samples".
          format(i, SignNames[i], np.where(y_train==i)[0].shape[0], np.where(y_te
st==i)[0].shape[0]) )
```

```
Number of training examples = 39209
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
Class 0 (Speed limit (20km/h)) has 210 training samples and 60 testing samples
Class 1 (Speed limit (30km/h)) has 2220 training samples and 720 testing samples
Class 2 (Speed limit (50km/h)) has 2250 training samples and 750 testing samples
Class 3 (Speed limit (60km/h)) has 1410 training samples and 450 testing samples
Class 4 (Speed limit (70km/h)) has 1980 training samples and 660 testing samples
Class 5 (Speed limit (80km/h)) has 1860 training samples and 630 testing samples
Class 6 (End of speed limit (80km/h)) has 420 training samples and 150 testing s
amples
Class 7 (Speed limit (100km/h)) has 1440 training samples and 450 testing sample
s
Class 8 (Speed limit (120km/h)) has 1410 training samples and 450 testing sample
s
Class 9 (No passing) has 1470 training samples and 480 testing samples
Class 10 (No passing for vehicles over 3.5 metric tons) has 2010 training sample
s and 660 testing samples
Class 11 (Right-of-way at the next intersection) has 1320 training samples and 4
20 testing samples
Class 12 (Priority road) has 2100 training samples and 690 testing samples
Class 13 (Yield) has 2160 training samples and 720 testing samples
Class 14 (Stop) has 780 training samples and 270 testing samples
Class 15 (No vehicles) has 630 training samples and 210 testing samples
Class 16 (Vehicles over 3.5 metric tons prohibited) has 420 training samples and
 150 testing samples
Class 17 (No entry) has 1110 training samples and 360 testing samples
Class 18 (General caution) has 1200 training samples and 390 testing samples
Class 19 (Dangerous curve to the left) has 210 training samples and 60 testing s
amples
Class 20 (Dangerous curve to the right) has 360 training samples and 90 testing
 samples
Class 21 (Double curve) has 330 training samples and 90 testing samples
Class 22 (Bumpy road) has 390 training samples and 120 testing samples
Class 23 (Slippery road) has 510 training samples and 150 testing samples
Class 24 (Road narrows on the right) has 270 training samples and 90 testing sam
ples
Class 25 (Road work) has 1500 training samples and 480 testing samples
Class 26 (Traffic signals) has 600 training samples and 180 testing samples
Class 27 (Pedestrians) has 240 training samples and 60 testing samples
Class 28 (Children crossing) has 540 training samples and 150 testing samples
Class 29 (Bicycles crossing) has 270 training samples and 90 testing samples
Class 30 (Beware of ice/snow) has 450 training samples and 150 testing samples
Class 31 (Wild animals crossing) has 780 training samples and 270 testing sample
s
Class 32 (End of all speed and passing limits) has 240 training samples and 60 t
esting samples
Class 33 (Turn right ahead) has 689 training samples and 210 testing samples
Class 34 (Turn left ahead) has 420 training samples and 120 testing samples
Class 35 (Ahead only) has 1200 training samples and 390 testing samples
Class 36 (Go straight or right) has 390 training samples and 120 testing samples
Class 37 (Go straight or left) has 210 training samples and 60 testing samples
Class 38 (Keep right) has 2070 training samples and 690 testing samples
Class 39 (Keep left) has 300 training samples and 90 testing samples
Class 40 (Roundabout mandatory) has 360 training samples and 90 testing samples
Class 41 (End of no passing) has 240 training samples and 60 testing samples
Class 42 (End of no passing by vehicles over 3.5 metric tons) has 240 training s
amples and 90 testing samples
```

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/)](http://matplotlib.org/) [examples (http://matplotlib.org/examples/index.html)](http://matplotlib.org/examples/index.html) and [gallery (http://matplotlib.org/gallery.html)](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 [89]:

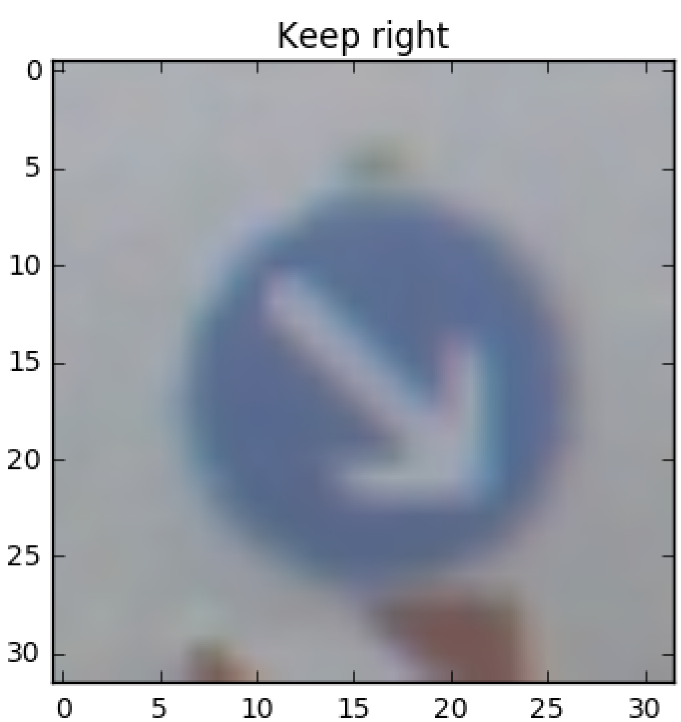
```
def show_example_sign(X, y, signClass):
    img = None
    indexList = np.where(y==signClass)[0]
    k = random.randint(0, indexList.size)
    im = X[indexList[k],:,:,:]
    #print(signClass, k)
    #plt.figure(figsize=(1,1))
    img = plt.imshow(im)
    plt.draw()
```

In [134]:

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

# Visualizations will be shown in the notebook.
%matplotlib inline
signClass = 38
show_example_sign(X_train, y_train, signClass)
plt.title(SignNames[signClass])
```

Out[134]:

```
<matplotlib.text.Text at 0x7f7b6d960e10>
```



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!

Implementation

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

In [7]:
```
def gaussian_filter(kernel_shape):
    x = np.zeros(kernel_shape, dtype=float)
    def gauss(x, y, sigma=2.0):
        Z = 2 * np.pi * sigma ** 2
        return  1. / Z * np.exp(-(x ** 2 + y ** 2) / (2. * sigma ** 2))
    mid = np.floor(kernel_shape[0] / 2.)
    for i in range(0, kernel_shape[0]):
        for j in range(0, kernel_shape[1]):
                x[i, j] = gauss(i - mid, j - mid)

    return x / np.sum(x)

# Local contrast normalization
def lecun_lcn(input, kernel_shape, threshold=1e-4):
    padSize = int(np.floor(kernel_shape / 2.))
    inputMin = np.min(input)
    padInput = cv2.copyMakeBorder(input, padSize, padSize, padSize, padSize,
cv2.BORDER_CONSTANT,value = 0)
    convout =  cv2.GaussianBlur(padInput,(kernel_shape, kernel_shape),0)
    centered_X = input - convout[padSize:-padSize, padSize:-padSize]
    centered_Xmin = np.min(centered_X)
    sum_sq_XX = cv2.GaussianBlur(centered_X**2,(kernel_shape, kernel_shape),0)
    pad_sum_sq_XX = cv2.copyMakeBorder(sum_sq_XX, padSize, padSize, padSize, padS
ize,
                                   cv2.BORDER_CONSTANT,value = 0)
    denorm = np.sqrt(pad_sum_sq_XX[padSize:-padSize, padSize:-padSize])
    #img_mean = np.mean(denorm)
    #divisor = np.maximum(denorm, img_mean)
    divisor = np.maximum(denorm, threshold)

    #img_mean = np.mean(denorm)
    #divisor = np.maximum(, threshold)
    return centered_X/divisor

# Global contrast normalization
def gcn(X):
    X = X.astype(float)
    mean = np.mean(X)
    X = X - mean
    normalizer = np.sqrt(np.sum(X**2.0))
    return X / np.maximum(normalizer, 1e-8)
```

In [8]:

```
### Preprocess the data here.
### Feel free to use as many code cells as needed.

#for i in range(n_train):
#X_train_copy = np.zeros_like(X_train,dtype=float)
#for i in range(n_train):
#    X_train_copy[i,:,:,:] = cv2.cvtColor(X_train[i,:,:,:], cv2.COLOR_RGB2YUV)
#    Ychannel = X_train_copy[i,:,:,0]
#    # Global contrast normalization
#    Ymean = np.mean(Ychannel)
#    Ychannel = Ychannel - Ymean
    # Local contrast normalization
#    Ylcn = lecun_lcn(Ychannel,9)
#    X_train_copy[i,:,:,0] = Ylcn
    #plt.imshow(Ylcn,cmap = 'Greys_r' )
#X_test_copy = np.zeros_like(X_test,dtype=float)
#for i in range(n_test):
#    X_test_copy[i,:,:,:] = cv2.cvtColor(X_test[i,:,:,:], cv2.COLOR_RGB2YUV)
#    Ychannel = X_test_copy[i,:,:,0]
    # Global contrast normalization
#    Ymean = np.mean(Ychannel)
#    Ychannel = Ychannel - Ymean
#    # Local contrast normalization
#    Ylcn = lecun_lcn(Ychannel,3)
#    X_test_copy[i,:,:,0] = Ylcn
```

In [9]:
```
### Preprocess the data here.

# Convert the color image to gray image and apply the global constrast normalizat
ion
X_train_gray = np.zeros(X_train.shape[0:3], dtype=float)
X_test_gray = np.zeros(X_test.shape[0:3], dtype=float)
for i in range(n_train):
    img = cv2.cvtColor(X_train[i,:,:,:], cv2.COLOR_RGB2GRAY)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = gcn(img)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = lecun_lcn(img, 3)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    X_train_gray[i,] = img
for i in range(n_test):
    img = gcn(cv2.cvtColor(X_test[i,:,:,:], cv2.COLOR_RGB2GRAY))
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = gcn(img)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = lecun_lcn(img, 3)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    X_test_gray[i,] = img

X_train_gray = X_train_gray.reshape((n_train,32,32,1))
X_test_gray = X_test_gray.reshape((n_test,32,32,1))
y_train_gray = y_train
```

Question 1

Describe how you preprocessed the data. Why did you choose that technique?

Answer: As stated in the paper "Traffic Sign Recoginiton with Multi-Scale Convolutional Networks", ignoring the color information had achieved the best test accuracy. Thus, I preprocessed the data by converting the color images to be gray images. In addition, I did some experiments on running the NN on color images but the test accuracy is not improved. In addition, I applied global constrast and local contrast normalization on gray scale image.

In [10]:

```
### Generate data additional data (OPTIONAL!)
### and split the data into training/validation/testing sets here.
### Feel free to use as many code cells as needed.

def shift_image(img, left_range, right_range):
    shift_left = random.randint(left_range[0], left_range[1])
    shift_right = random.randint(right_range[0], right_range[1])
    rows,cols = img.shape[0:2]
    M = np.float32([[1,0,shift_left],[0,1,shift_right]])
    for i in range(img.shape[2]):
        img[:,:,i] =  cv2.warpAffine(img[:,:,i],M,(cols,rows))
    return img

def scale_image(img, scale_range):
    scale = random.uniform(scale_range[0], scale_range[1])
    for i in range(img.shape[2]):
        orig = img[:,:,i]
        res = cv2.resize(orig,None,fx=scale, fy=scale, interpolation = cv2.INTER_
CUBIC)
        shift_x = int(abs(res.shape[0] - img.shape[0])/2)
        shift_y = int(abs(res.shape[1] - img.shape[1])/2)
        tmp = np.ones_like(orig, dtype = float) * np.min(orig)
        if scale > 1:
            tmp = res[shift_x:(shift_x+img.shape[0]), shift_y:
(shift_y+img.shape[1])]
        else:
            tmp[shift_x:(shift_x+res.shape[0]), shift_y:(shift_y+res.shape[1])] =
 res
        img[:,:,i] = tmp
    return img

def rotate_image(img, rotate_range):
    rotate_angle = random.randint(rotate_range[0], rotate_range[1])
    rows,cols = img.shape[0:2]
    M = cv2.getRotationMatrix2D((cols/2,rows/2),rotate_angle,1)
    return cv2.warpAffine(img,M,(cols,rows))
```

In [11]:

```
#X_train_jittered, y_train_jittered = shuffle(X_train_jittered, y_train_jittered)
#X_validation = X_train_jittered[0:8000,]
#y_validation = y_train_jittered[0:8000,]
#X_train = X_train_jittered[8000:-1,]
#y_train = y_train_jittered[8000:-1,]

X_train_copy = X_train_gray.copy()
y_train_copy = y_train_gray.copy()
X_train_copy, y_train_copy = shuffle(X_train_copy, y_train_copy)
X_validation = np.empty((0,32,32,1), dtype=float)
y_validation = np.empty((0), dtype=float)
X_train = np.empty((0,32,32,1), dtype=float)
y_train = np.empty((0), dtype=float)
for signClass in range(n_classes):
    ind = np.where(y_train_copy == signClass)[0]
    y_validation = np.append(y_validation, y_train_copy[ind[0:50],], axis =0)
    X_validation = np.append(X_validation, X_train_copy[ind[0:50],], axis =0)
    y_train = np.append(y_train, y_train_copy[ind[50:-1],], axis =0)
    X_train = np.append(X_train, X_train_copy[ind[50:-1],], axis =0)
```

In [ ]:
```
numOfTimesForJitteredData = 1
y_train_jittered = y_train.copy()
for i in range(numOfTimesForJitteredData):
    y_train_jittered = np.concatenate((y_train_jittered, y_train))
print(y_train_jittered.shape)
X_train_jittered = np.empty((0,32,32,1), dtype=float)
for i in range(X_train.shape[0]*numOfTimesForJitteredData):
    tmp = X_train.copy()[i%X_train.shape[0],:,:,:]
    tmp = shift_image(tmp,[-2,2],[-2,2])
    tmp = scale_image(tmp,[0.9, 1.1])
    tmp = rotate_image(tmp,[-15, 15])
    X_train_jittered = np.append(X_train_jittered, tmp.reshape(1,32,32,1),
axis=0)
    if i%1000 == 0:
        print(i)
X_train_jittered = np.concatenate((X_train_jittered, X_train))
```

In [20]:
```
#np.save("jitteredX", X_train_jittered)
#np.save("jitteredy", y_train_jittered)
```

In [12]:
```
#X_train_jittered = np.load("jitteredX.npy")
#y_train_jittered = np.load("jitteredy.npy")
```

Question 2

*Describe how you set up the training, validation and testing data for your model. **Optional**: If you generated additional data, how did you generate the data? Why did you generate the data? What are the differences in the new dataset (with generated data) from the original dataset?*

Answer: I keep the original testing data as test data. For each class of sign, I sampled 50 images from the training data to form the validation data and use the rest of them as the new training data. I also jittered data by randomly shifting image by [-2, 2] pixels in both direction, scaling image by [0.9,1.1], and rotating image by [-15, 15] degrees. After jittering, the total training size is doubled. I think by jittering the data, my model will be more robust. This has been confirmed in the later experiments.

In [13]:

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

from tensorflow.contrib.layers import flatten

def LeNetLab(x, n_classes, channel):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # TODO: Layer 1: Convolutional. Input = 32x32x1. Output = 32x32x6.
    W_conv1 = tf.Variable(tf.truncated_normal([5, 5, channel, 6], mean = mu, stdd
ev = sigma), name='W1')
    b_conv1 = tf.Variable(tf.zeros(6), name='b1')
    conv1 = tf.nn.conv2d(x, W_conv1, strides=[1, 1, 1, 1], padding='VALID') + b_c
onv1

    # TODO: Activation.
    conv1 = tf.nn.relu(conv1)

    # TODO: Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='VALID')

    # TODO: Layer 2: Convolutional. Output = 10x10x16.
    W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 6, 16], mean = mu, stddev =
sigma), name='W2')
    b_conv2 = tf.Variable(tf.zeros(16), name='b2')
    conv2 = tf.nn.conv2d(conv1, W_conv2, strides=[1, 1, 1, 1], padding='VALID') +
 b_conv2

    # TODO: Activation.
    conv2 = tf.nn.relu(conv2)

    # TODO: Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='VALID')

    # TODO: Flatten. Input = 5x5x16. Output = 400.
    fc0 = flatten(conv2)

    # TODO: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal([400, 120], mean = mu, stddev =
sigma), name='W3')
    fc1_b = tf.Variable(tf.zeros(120), name='b3')
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # TODO: Activation.
    fc1 = tf.nn.relu(fc1)

    # TODO: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal([120, 84], mean = mu, stddev =
sigma), name='W4')
    fc2_b = tf.Variable(tf.zeros(84), name='b4')
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # TODO: Activation.
    fc2 = tf.nn.relu(fc2)

    # TODO: Layer 5: Fully Connected. Input = 84. Output = n_classses.
    fc3_W = tf.Variable(tf.truncated_normal([84, n_classes], mean = mu, stddev =
sigma), name='W5')
```

```
    fc3_b = tf.Variable(tf.zeros(n_classes), name='b5')
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

In [13]:

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

from tensorflow.contrib.layers import flatten

def LeNetLabMid(x, n_classes, channel):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # TODO: Layer 1: Convolutional. Input = 32x32x1. Output = 32x32x16.
    W_conv1 = tf.Variable(tf.truncated_normal([5, 5, channel, 16], mean = mu, std
dev = sigma))
    b_conv1 = tf.Variable(tf.zeros(16))
    conv1 = tf.nn.conv2d(x, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_co
nv1

    # TODO: Activation.
    conv1 = tf.nn.relu(conv1)

    # TODO: Pooling. Input = 32x32x16. Output = 16x16x16.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='SAME')

    # TODO: Layer 2: Convolutional. Output = 16x16x32.
    W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 16, 32], mean = mu, stddev =
 sigma))
    b_conv2 = tf.Variable(tf.zeros(32))
    conv2 = tf.nn.conv2d(conv1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') +
b_conv2

    # TODO: Activation.
    conv2 = tf.nn.relu(conv2)

    # TODO: Pooling. Input = 16x16x32. Output = 8x8x32.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='SAME')

    # TODO: Flatten. Input = 8x8x32. Output = 2048.
    fc0 = flatten(conv2)

    # TODO: Layer 3: Fully Connected. Input = 2048. Output = 512.
    fc1_W = tf.Variable(tf.truncated_normal([2048, 512], mean = mu, stddev = sigm
a))
    fc1_b = tf.Variable(tf.zeros(512))
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # TODO: Activation.
    fc1 = tf.nn.relu(fc1)

    # TODO: Layer 4: Fully Connected. Input = 512. Output = 128.
    fc2_W = tf.Variable(tf.truncated_normal([512, 128], mean = mu, stddev =
sigma))
    fc2_b = tf.Variable(tf.zeros(128))
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # TODO: Activation.
    fc2 = tf.nn.relu(fc2)

    # TODO: Layer 5: Fully Connected. Input = 128. Output = n_classses.
    fc3_W = tf.Variable(tf.truncated_normal([128, n_classes], mean = mu, stddev =
 sigma))
```

```
    fc3_b = tf.Variable(tf.zeros(n_classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

In [15]:

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

from tensorflow.contrib.layers import flatten

def LeNet(x, n_classes, channel):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # TODO: Layer 1: Convolutional. Input = 32x32x1. Output = 32x32x108.
    W_conv1 = tf.Variable(tf.truncated_normal([5, 5, channel, 108], mean = mu, st
ddev = sigma), name='W1')
    b_conv1 = tf.Variable(tf.zeros(108), name='b1')
    conv1 = tf.nn.conv2d(x, W_conv1, strides=[1, 1, 1, 1], padding='SAME') + b_co
nv1

    # TODO: Activation.
    conv1 = tf.nn.relu(conv1)

    # TODO: Pooling. Input = 32x32x108. Output = 16x16x108.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='SAME')

    # TODO: Layer 2: Convolutional. Output = 16x16x108.
    W_conv2 = tf.Variable(tf.truncated_normal([5, 5, 108, 108], mean = mu, stddev
 = sigma), name='W2')
    b_conv2 = tf.Variable(tf.zeros(108), name='b2')
    conv2 = tf.nn.conv2d(conv1, W_conv2, strides=[1, 1, 1, 1], padding='SAME') +
b_conv2

    # TODO: Activation.
    conv2 = tf.nn.relu(conv2)

    # TODO: Pooling. Input = 16x16x108. Output = 8x8x108.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='VALID')

    # TODO: Flatten. Input = 8x8x108. Output = 6912.
    fc0 = flatten(conv2)

    # TODO: Layer 3: Fully Connected. Input = 6912. Output = 2000.
    fc1_W = tf.Variable(tf.truncated_normal([6912, 2000], mean = mu, stddev = sig
ma), name='W3')
    fc1_b = tf.Variable(tf.zeros(2000), name='b3')
    fc1 = tf.matmul(fc0, fc1_W) + fc1_b

    # TODO: Activation.
    fc1 = tf.nn.relu(fc1)

    # TODO: Layer 4: Fully Connected. Input = 2000. Output = 500.
    fc2_W = tf.Variable(tf.truncated_normal([2000, 500], mean = mu, stddev = sigm
a), name='W4')
    fc2_b = tf.Variable(tf.zeros(500), name='b4')
    fc2 = tf.matmul(fc1, fc2_W) + fc2_b

    # TODO: Activation.
    fc2 = tf.nn.relu(fc2)

    # TODO: Layer 5: Fully Connected. Input = 500. Output = n_classses.
    fc3_W = tf.Variable(tf.truncated_normal([500, n_classes], mean = mu, stddev =
 sigma), name='W5')
```

```python
    fc3_b = tf.Variable(tf.zeros(n_classes), name='b5')
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

In [16]:

```

```

```
def VGG16(x, n_classes, channel):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # Conv Layer 1_1: Convolutional. Input = 32x32xchannel. Output = 32x32x64.
    kernel1_1 = tf.Variable(tf.truncated_normal([3, 3, channel, 64], dtype=tf.flo
at32,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(x, kernel1_1, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv1_1 = tf.nn.relu(out)

    # Conv Layer 1_2: Convolutional. Input = 32x32x64. Output = 32x32x64.
    kernel1_2 = tf.Variable(tf.truncated_normal([3, 3, 64, 64], dtype=tf.float32,
                                                  stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(conv1_1, kernel1_2, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                                 trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv1_2 = tf.nn.relu(out)

    # pooling
    pool1 = tf.nn.max_pool(conv1_2,
                        ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME',
                        name='pool1')

    # Conv Layer 2_1: Convolutional. Input = 16x16x64. Output = 16x16x128.
    kernel2_1 = tf.Variable(tf.truncated_normal([3, 3, 64, 128],
dtype=tf.float32,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(pool1, kernel2_1, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[128], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv2_1 = tf.nn.relu(out)

    # Conv Layer 2_2: Convolutional. Input = 16x16x128. Output = 16x16x128.
    kernel2_2 = tf.Variable(tf.truncated_normal([3, 3, 128, 128], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(conv2_1, kernel2_2, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[128], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv2_2 = tf.nn.relu(out,)

    # pooling
    pool2 = tf.nn.max_pool(conv2_2,
                        ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME',
                        name='pool2')

     # Conv Layer 3_1: Convolutional. Input = 8x8x128. Output = 8x8x256.
```

```
    kernel3_1 = tf.Variable(tf.truncated_normal([3, 3, 128, 256], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(pool2, kernel3_1, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv3_1 = tf.nn.relu(out)

    # Conv Layer 3_2: Convolutional. Input = 8x8x256. Output = 8x8x256.
    kernel3_2 = tf.Variable(tf.truncated_normal([3, 3, 256, 256], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(conv3_1, kernel3_2, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv3_2 = tf.nn.relu(out)

    # Conv Layer 3_3: Convolutional. Input = 8x8x256. Output = 8x8x256.
    kernel3_3 = tf.Variable(tf.truncated_normal([3, 3, 256, 256], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(conv3_2, kernel3_3, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv3_3 = tf.nn.relu(out)

    # pooling
    pool3 = tf.nn.max_pool(conv3_3,
                        ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1],
                        padding='SAME',
                        name='pool3')


    # Conv Layer 4_1: Convolutional. Input = 4x4x256. Output = 4x4x512.
    kernel4_1 = tf.Variable(tf.truncated_normal([3, 3, 256, 512], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(pool3, kernel4_1, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[512], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv4_1 = tf.nn.relu(out)

    # Conv Layer 4_2: Convolutional. Input = 4x4x512. Output = 4x4x512.
    kernel4_2 = tf.Variable(tf.truncated_normal([3, 3, 512, 512], dtype=tf.float3
2,
                                                          stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(conv4_1, kernel4_2, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[512], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv4_2 = tf.nn.relu(out)
```

```
    # Conv Layer 4_3: Convolutional. Input = 4x4x512. Output = 4x4x512.
    kernel4_3 = tf.Variable(tf.truncated_normal([3, 3, 512, 512], dtype=tf.float3
2,
                                                     stddev=1e-1),
name='weights')
    conv = tf.nn.conv2d(conv4_2, kernel4_3, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[512], dtype=tf.float32),
                         trainable=True, name='biases')
    out = tf.nn.bias_add(conv, biases)
    conv4_3 = tf.nn.relu(out)

    # pooling
    pool4 = tf.nn.max_pool(conv4_3,
                           ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1],
                           padding='SAME',
                           name='pool4')



    # Flatten. Input = 4x4x512. Output = 8192.
    fc0 = flatten(pool4)
    shape = int(np.prod(pool4.get_shape()[1:]))

     # Fully Connected 1. Input = 8192. Output = 4096.

    fc1w = tf.Variable(tf.truncated_normal([shape, 4096],
                                           dtype=tf.float32,
                                           stddev=1e-1), name='weights')
    fc1b = tf.Variable(tf.constant(1.0, shape=[4096], dtype=tf.float32),
                                 trainable=True, name='biases')
    fc1l = tf.nn.bias_add(tf.matmul(fc0, fc1w), fc1b)
    fc1 = tf.nn.relu(fc1l)

    # Fully Connected 2. Input = 4096. Output = 4096.
    fc2w = tf.Variable(tf.truncated_normal([4096, 4096],
                                           dtype=tf.float32,
                                           stddev=1e-1), name='weights')
    fc2b = tf.Variable(tf.constant(1.0, shape=[4096], dtype=tf.float32),
                                 trainable=True, name='biases')
    fc2l = tf.nn.bias_add(tf.matmul(fc1, fc2w), fc2b)
    fc2 = tf.nn.relu(fc2l)

    # Fully Connected 2. Input = 4096. Output = n_classes.
    fc3w = tf.Variable(tf.truncated_normal([4096, n_classes],
                                           dtype=tf.float32,
                                           stddev=1e-1), name='weights')
    fc3b = tf.Variable(tf.constant(1.0, shape=[n_classes], dtype=tf.float32),
                                 trainable=True, name='biases')
    logits = tf.nn.bias_add(tf.matmul(fc2, fc3w), fc3b)

    return logits
```

Question 3

What does your final architecture look like? (Type of model, layers, sizes, connectivity, etc.) For reference on how to build a deep neural network using TensorFlow, see [Deep Neural Network in TensorFlow (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/b516a270-8600-4f93-a0a3-20dfeabe5da6/concepts/83a3a2a2-a9bd-4b7b-95b0-eb924ab14432)](https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/b516a270-8600-4f93-a0a3-20dfeabe5da6/concepts/83a3a2a2-a9bd-4b7b-95b0-eb924ab14432) from the classroom.

Answer: I use LetNet network (see LeNetLab) in class with its original layers, sizes and connectivity first. Then I used larger size convolution in the first two layers and more connectivity in the last three fully connected layers （see LeNetLabMid and LeNet). In addition, I constructed VGG16 with same number of layers but smaller number of filter and less hidden units. The best result I obtained is based on LeNetLabMid which has five layers and the first layer have 32 convolution filters, second layer has 32 convolution filters, third layer fully connects to 512 hidden units, fourth layer fully connects to 256 hidden units, and last layer fully connects to 43 predictions.

In [15]:
```
### Train your model here.
### Feel free to use as many code cells as needed.
x = tf.placeholder(tf.float32, (None, 32, 32, channel))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)
channel = 1
```

In [16]:
```
logits = LeNetLabMid(x, n_classes, channel)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
```

In [17]:
```
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:offset+BATCH_SIZE]
        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 [18]:

```
best_EPOCHS = 0
best_accuracy = 0
best_BATCH_SIZE = 0
best_rate = 0;
X_train = X_train_jittered
y_train = y_train_jittered
X_test = X_test_gray
num_examples = len(X_train)
model_file = 'lenetlab_best.ckpt'
for rate in [0.001]: #[0.0005, 0.001, 0.002]:
    for BATCH_SIZE in [256]: #[128, 256, 512]:
        for EPOCHS in [100]: #[10, 50, 100]:
            optimizer = tf.train.AdamOptimizer(learning_rate = rate)
            training_operation = optimizer.minimize(loss_operation)
            print(rate,BATCH_SIZE,EPOCHS)
            with tf.Session() as sess:
                sess.run(tf.global_variables_initializer())
                print("Training...")
                print()
                for i in range(EPOCHS):
                    #X_train, y_train = shuffle(X_train_jittered.copy()[-int(num_
examples/2)-1:-1], y_train_jittered.copy()[-int(num_examples/2)-1:-1])
                    X_train, y_train = shuffle(X_train, y_train)
                   # print(X_train.shape)
                   # print(X_train_jittered.shape)
                    for offset in range(0, num_examples, BATCH_SIZE):
                        end = offset + BATCH_SIZE
                        batch_x, batch_y = X_train[offset:end], y_train[offset:en
d]

                        sess.run(training_operation, feed_dict={x: batch_x, y: ba
tch_y})
                    validation_accuracy = evaluate(X_validation, y_validation)
                    print("EPOCH {} ...".format(i+1))
                    print("Validation Accuracy = {:.3f}".format(validation_accura
cy))

                    print()
                  #   test_accuracy = evaluate(X_test, y_test)
                   #  print("Test Accuracy = {:.3f}".format(test_accuracy))
                if (validation_accuracy > best_accuracy) :
                    saver.save(sess, model_file)
                    best_EPOCHS = EPOCHS
                    best_accuracy = validation_accuracy
                    best_BATCHSIZE = BATCH_SIZE
                    best_rate = rate
                    test_accuracy = evaluate(X_test, y_test)
                    print("Test Accuracy = {:.3f}".format(test_accuracy))
                    print("Model saved")
```

```
0.001 256 100
Training...

EPOCH 1 ...
Validation Accuracy = 0.913

EPOCH 2 ...
Validation Accuracy = 0.976

EPOCH 3 ...
Validation Accuracy = 0.993

EPOCH 4 ...
Validation Accuracy = 0.990

EPOCH 5 ...
Validation Accuracy = 0.998

EPOCH 6 ...
Validation Accuracy = 0.989

EPOCH 7 ...
Validation Accuracy = 0.994

EPOCH 8 ...
Validation Accuracy = 0.997

EPOCH 9 ...
Validation Accuracy = 0.998

EPOCH 10 ...
Validation Accuracy = 0.994

EPOCH 11 ...
Validation Accuracy = 0.993

EPOCH 12 ...
Validation Accuracy = 0.996

EPOCH 13 ...
Validation Accuracy = 0.999

EPOCH 14 ...
Validation Accuracy = 1.000

EPOCH 15 ...
Validation Accuracy = 1.000

EPOCH 16 ...
Validation Accuracy = 0.996

EPOCH 17 ...
Validation Accuracy = 0.999

EPOCH 18 ...
Validation Accuracy = 0.997

EPOCH 19 ...
Validation Accuracy = 0.996

EPOCH 20 ...
Validation Accuracy = 0.999
```

```
EPOCH 21 ...
Validation Accuracy = 0.999

EPOCH 22 ...
Validation Accuracy = 0.997

EPOCH 23 ...
Validation Accuracy = 0.999

EPOCH 24 ...
Validation Accuracy = 0.999

EPOCH 25 ...
Validation Accuracy = 0.997

EPOCH 26 ...
Validation Accuracy = 1.000

EPOCH 27 ...
Validation Accuracy = 0.997

EPOCH 28 ...
Validation Accuracy = 0.999

EPOCH 29 ...
Validation Accuracy = 0.999

EPOCH 30 ...
Validation Accuracy = 0.997

EPOCH 31 ...
Validation Accuracy = 0.997

EPOCH 32 ...
Validation Accuracy = 1.000

EPOCH 33 ...
Validation Accuracy = 1.000

EPOCH 34 ...
Validation Accuracy = 0.999

EPOCH 35 ...
Validation Accuracy = 0.999

EPOCH 36 ...
Validation Accuracy = 1.000

EPOCH 37 ...
Validation Accuracy = 0.999

EPOCH 38 ...
Validation Accuracy = 1.000

EPOCH 39 ...
Validation Accuracy = 1.000

EPOCH 40 ...
Validation Accuracy = 0.999

EPOCH 41 ...
```

```
Validation Accuracy = 0.999

EPOCH 42 ...
Validation Accuracy = 0.999

EPOCH 43 ...
Validation Accuracy = 1.000

EPOCH 44 ...
Validation Accuracy = 1.000

EPOCH 45 ...
Validation Accuracy = 0.999

EPOCH 46 ...
Validation Accuracy = 0.999

EPOCH 47 ...
Validation Accuracy = 0.998

EPOCH 48 ...
Validation Accuracy = 0.999

EPOCH 49 ...
Validation Accuracy = 0.998

EPOCH 50 ...
Validation Accuracy = 0.998

EPOCH 51 ...
Validation Accuracy = 1.000

EPOCH 52 ...
Validation Accuracy = 0.999

EPOCH 53 ...
Validation Accuracy = 1.000

EPOCH 54 ...
Validation Accuracy = 1.000

EPOCH 55 ...
Validation Accuracy = 1.000

EPOCH 56 ...
Validation Accuracy = 1.000

EPOCH 57 ...
Validation Accuracy = 1.000

EPOCH 58 ...
Validation Accuracy = 1.000

EPOCH 59 ...
Validation Accuracy = 1.000

EPOCH 60 ...
Validation Accuracy = 1.000

EPOCH 61 ...
Validation Accuracy = 1.000
```

```
EPOCH 62 ...
Validation Accuracy = 1.000

EPOCH 63 ...
Validation Accuracy = 1.000

EPOCH 64 ...
Validation Accuracy = 1.000

EPOCH 65 ...
Validation Accuracy = 1.000

EPOCH 66 ...
Validation Accuracy = 1.000

EPOCH 67 ...
Validation Accuracy = 1.000

EPOCH 68 ...
Validation Accuracy = 1.000

EPOCH 69 ...
Validation Accuracy = 1.000

EPOCH 70 ...
Validation Accuracy = 1.000

EPOCH 71 ...
Validation Accuracy = 1.000

EPOCH 72 ...
Validation Accuracy = 1.000

EPOCH 73 ...
Validation Accuracy = 1.000

EPOCH 74 ...
Validation Accuracy = 1.000

EPOCH 75 ...
Validation Accuracy = 1.000

EPOCH 76 ...
Validation Accuracy = 1.000

EPOCH 77 ...
Validation Accuracy = 1.000

EPOCH 78 ...
Validation Accuracy = 1.000

EPOCH 79 ...
Validation Accuracy = 1.000

EPOCH 80 ...
Validation Accuracy = 1.000

EPOCH 81 ...
Validation Accuracy = 1.000

EPOCH 82 ...
Validation Accuracy = 1.000
```

```
EPOCH 83 ...
Validation Accuracy = 1.000

EPOCH 84 ...
Validation Accuracy = 1.000

EPOCH 85 ...
Validation Accuracy = 1.000

EPOCH 86 ...
Validation Accuracy = 1.000

EPOCH 87 ...
Validation Accuracy = 1.000

EPOCH 88 ...
Validation Accuracy = 1.000

EPOCH 89 ...
Validation Accuracy = 1.000

EPOCH 90 ...
Validation Accuracy = 1.000

EPOCH 91 ...
Validation Accuracy = 1.000

EPOCH 92 ...
Validation Accuracy = 1.000

EPOCH 93 ...
Validation Accuracy = 1.000

EPOCH 94 ...
Validation Accuracy = 1.000

EPOCH 95 ...
Validation Accuracy = 1.000

EPOCH 96 ...
Validation Accuracy = 1.000

EPOCH 97 ...
Validation Accuracy = 1.000

EPOCH 98 ...
Validation Accuracy = 1.000

EPOCH 99 ...
Validation Accuracy = 1.000

EPOCH 100 ...
Validation Accuracy = 1.000

Test Accuracy = 0.971
Model saved
```

In [40]:
```
X_test = X_test_gray
print("Best EPOCHS: {} Best BATCHSIZE: {} Best rate: {} Best Validation Accuracy {}:".format(best_EPOCHS, best_BATCHSIZE,

      best_rate, best_accuracy))
BATCH_SIZE = 256
with tf.Session() as sess:
    #new_saver = tf.train.import_meta_graph(modelfile+'.meta')
    #new_saver.restore(sess, tf.train.latest_checkpoint('./'))
    saver.restore(sess, tf.train.latest_checkpoint('./'))
    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Best EPOCHS: 100 Best BATCHSIZE: 256 Best rate: 0.001 Best Validation Accuracy
 1.0:
Test Accuracy = 0.971
```

Question 4

How did you train your model? (Type of optimizer, batch size, epochs, hyperparameters, etc.)

Answer: I used AdamOptimizer and searched batch size in 128, 256, 512, epochs in 10, 50, 100 and learning rate in 0.0005, 0.001, 0.002. I choose the best settings based on best validation set accuracy.

Question 5

What approach did you take in coming up with a solution to this problem? It may have been a process of trial and error, in which case, outline the steps you took to get to the final solution and why you chose those steps. Perhaps your solution involved an already well known implementation or architecture. In this case, discuss why you think this is suitable for the current problem.

Answer: First, I tried different processing technique with original LeNet setting. Based on original RGB channel data, I got 90.8% test accuracy. Then I used YUV channels with same setting and got 86.1% test accuracy. By converting color image to Gray image, I got 93.8% test accuracy. After Global and local contrast normalization on gray image, the test accuracy is improved into 94.6%. Then I decided to use gray image with global and local contrast normalization as preprocessing. Then I trained the original data and the jittered data with different networks I described in answer for question 3 with hyperparameter setting I described in answer for question 4. Ther following is part of my experimental results.

```
        Network: LeLetLab
        original data test accuracy 95.7%
        Best EPOCHS: 100 Best BATCHSIZE: 256 Best rate: 0.002 Best Validation Accurac
y 0.9902325581395349:
        Test Accuracy = 0.957

        Best EPOCHS: 100 Best BATCHSIZE: 512 Best rate: 0.002 Best Validation Accurac
y 1.0:
        Test Accuracy = 0.960

        Network: LeNetLabMid
        jittered data test accuracy 97.1%
        Best EPOCHS: 100 Best BATCHSIZE: 256 Best rate: 0.001 Best Validation Accurac
y 1.0:
        Test Accuracy = 0.971

        original data test accuracy 95.9%
        Best EPOCHS: 100 Best BATCHSIZE: 128 Best rate: 0.001 Best Validation Accurac
y 0.9897674418604652:
        Test Accuracy = 0.959
```

We can see the jittered data always produce better results than the original data. In addition, LeNetLabMid produces better results than LeLetLab. The best test accuracy I got was 97.1%

Step 3: Test a Model on New Images

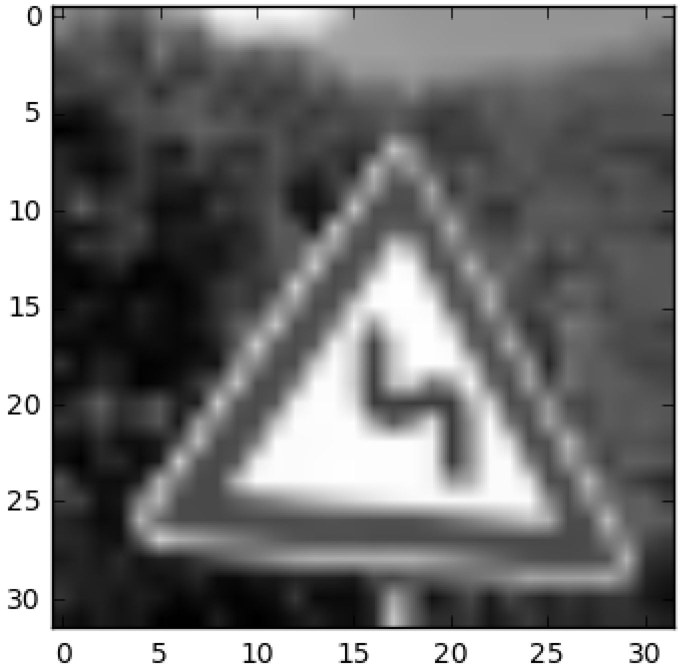
Take several pictures of traffic signs that you find on the web or around you (at least five), and run them through your classifier on your computer to produce example results. The classifier might not recognize some local signs but it could prove interesting nonetheless.

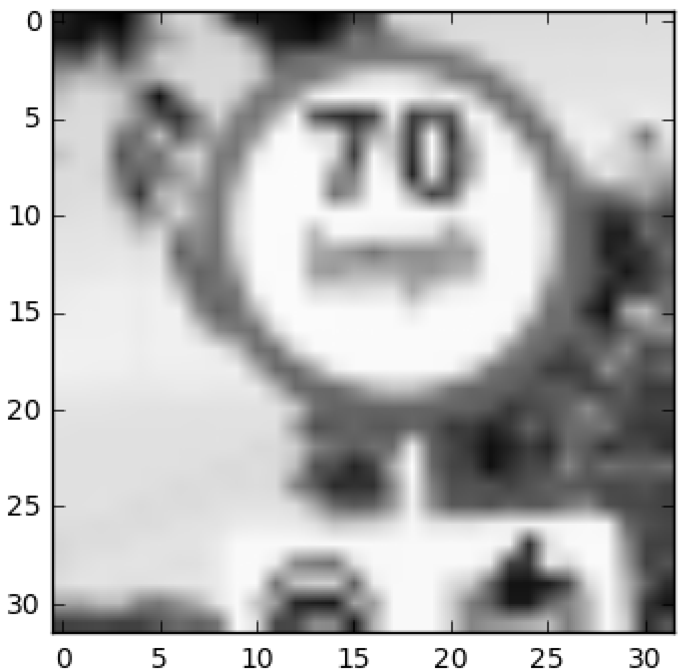
You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

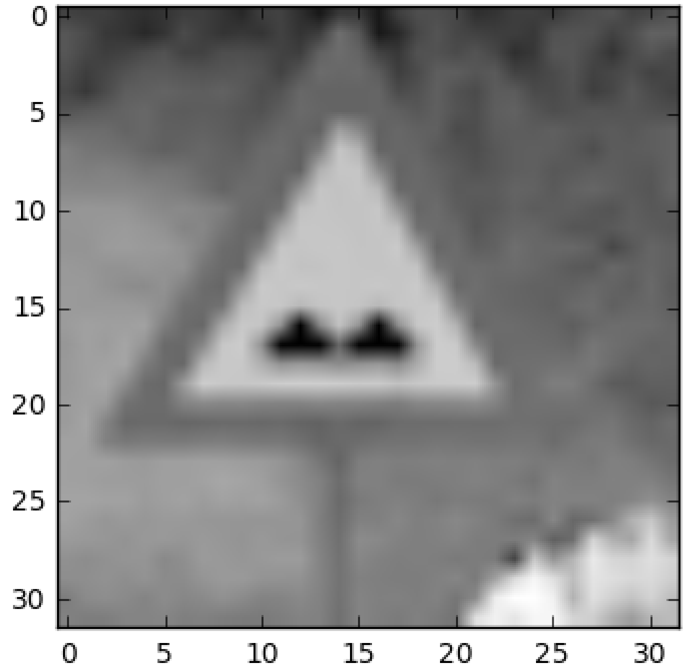
Implementation

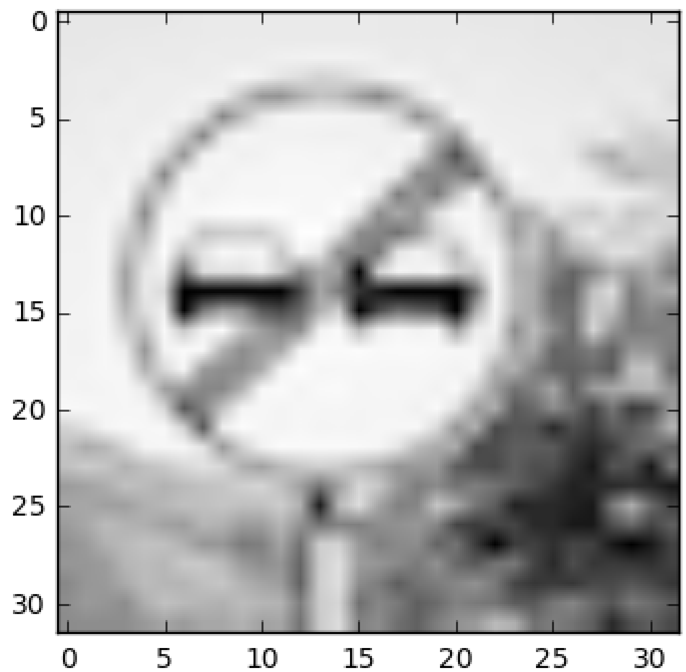
Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

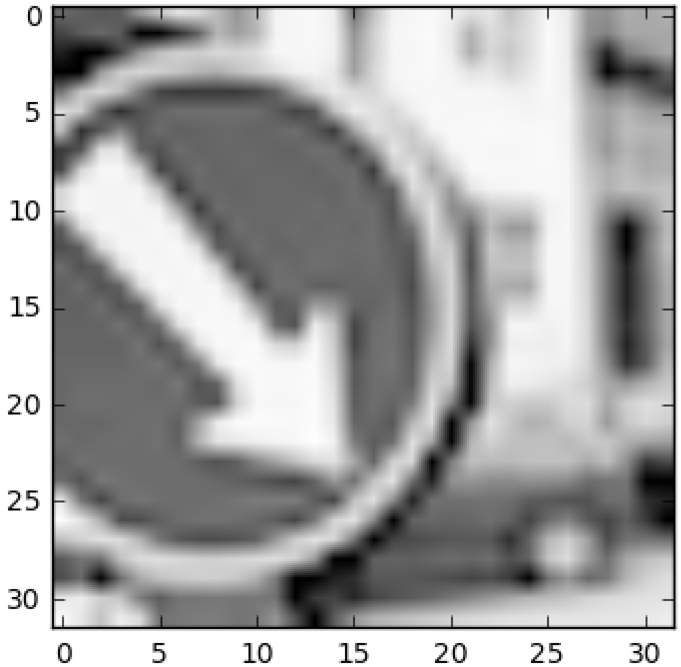
In [198]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
n_real = 5
for i in range(n_real):
    image = cv2.resize(cv2.imread('data/trafficSign/' + str(i+1) +'.png', 0),
(32,32), interpolation = cv2.INTER_AREA)
   # image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.imshow(image, cmap="Greys_r")
    plt.show()
```


0
5
10
15
20
25
30
0
5
10
15
20
25
30


70
0
5
10
15
20
25
30
0
5
10
15
20
25
30


0
5
10
15
20
25
30
0 5 10 15 20 25 30


0
5
10
15
20
25
30
0 5 10 15 20 25 30



Question 6

Choose five candidate images of traffic signs and provide them in the report. Are there any particular qualities of the image(s) that might make classification difficult? It could be helpful to plot the images in the notebook.

Answer: As shown in the notebook, the second and last images are quite challenge. The second image is 70 km speed limit sign but the number "70" is smaller than usual one because the text "kmph" below. The last image is not centered and only partial sign is in the image.

In [199]:

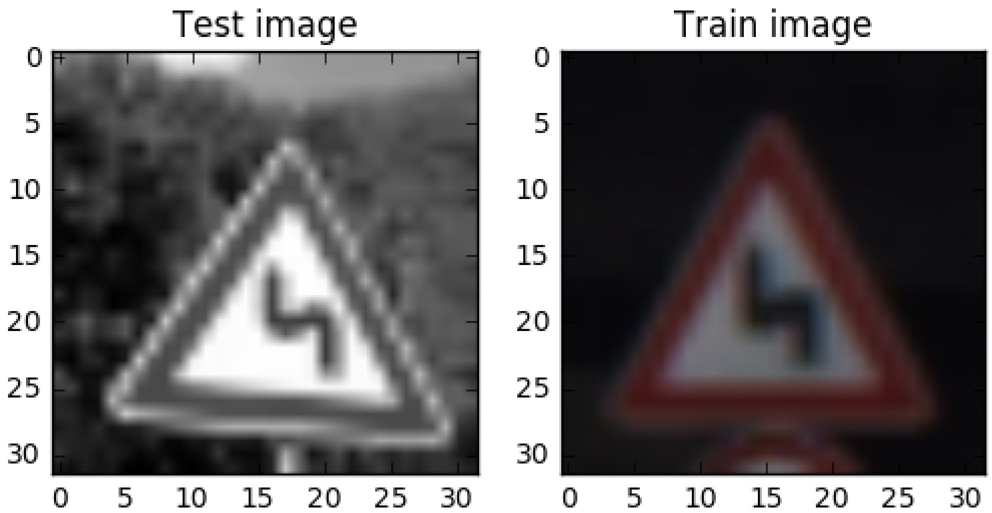
```
### Run the predictions here.
### Feel free to use as many code cells as needed.
X_real = np.empty((0,32,32,1), dtype=float)
for i in range(n_real):
    image = cv2.resize(cv2.imread('data/trafficSign/' + str(i+1) +'.png', 0),
(32,32), interpolation = cv2.INTER_AREA)
    X_real = np.append(X_real, image.reshape((1,32,32,1)), axis =0)
print(X_real.shape)
X_gray_real = np.zeros(X_real.shape[0:3], dtype=float)
for i in range(n_real):
    img = gcn(X_real[i,:,:,0])
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = gcn(img)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    img = lecun_lcn(img, 3)
    #plt.figure()
    #plt.imshow(img,cmap = 'Greys_r' )
    X_gray_real[i,] = img
X_gray_real = X_gray_real.reshape((n_real,32,32,1))

prediction = tf.argmax(logits, 1)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predicts = sess.run(prediction, feed_dict={x: X_gray_real})
```

```
(5, 32, 32, 1)
```

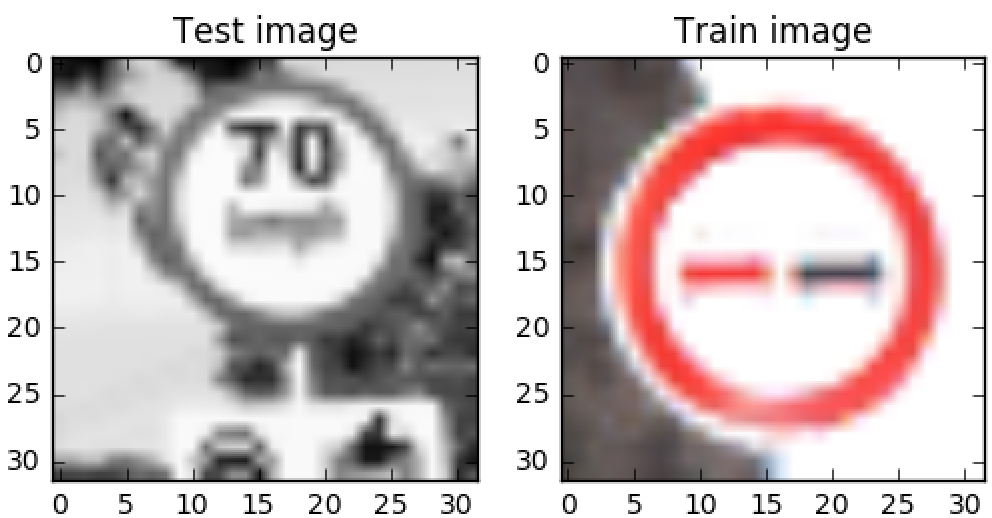
In [200]:

```
for i in range(n_real):
    signClass = predicts[i]
    plt.figure()
    plt.subplot(121)
    image = cv2.resize(cv2.imread('data/trafficSign/' + str(i+1) +'.png', 0),
(32,32), interpolation = cv2.INTER_AREA)
    plt.imshow(image, cmap="Greys_r")
    plt.title("Test image")
    #plt.imshow(X_gray_real[i,:,:,0], cmap="Greys_r")
    plt.subplot(122)
    show_example_sign(X_train, y_train, signClass)
    plt.title("Train image")
    plt.suptitle("The prediction of image {} is {}".format(i+1,SignNames[signClas
s]))
```

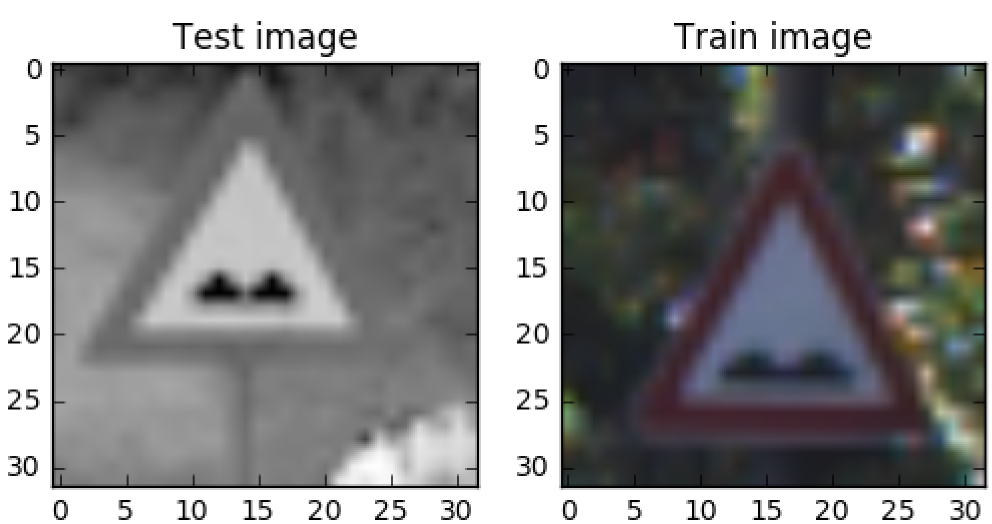
The prediction of image 1 is Double curve



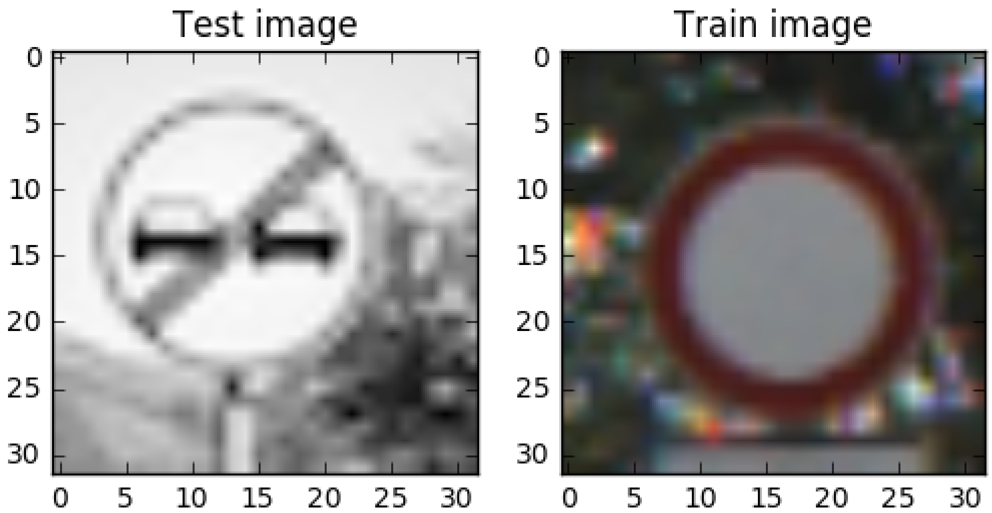
The prediction of image 2 is No passing



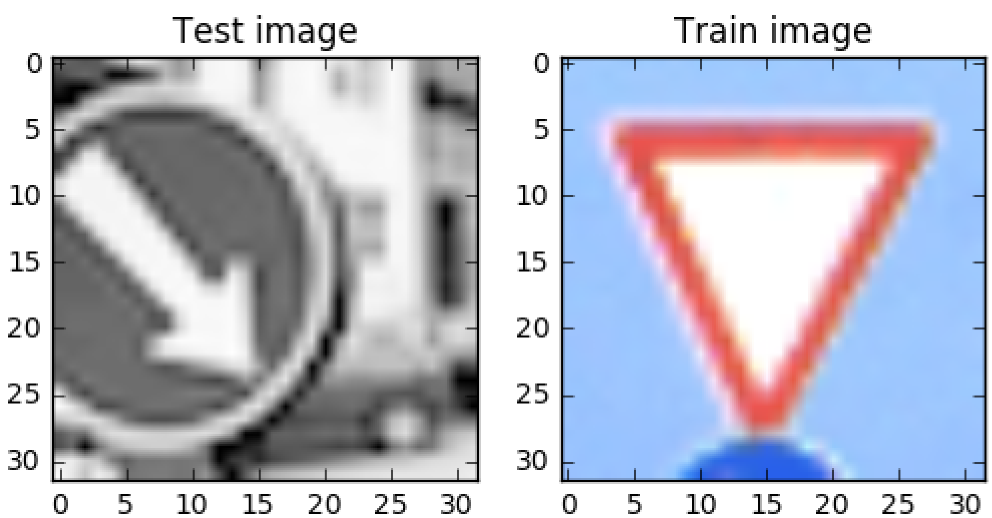
The prediction of image 3 is Bumpy road



The prediction of image 4 is No vehicles



The prediction of image 5 is Yield



Question 7

Is your model able to perform equally well on captured pictures when compared to testing on the dataset? The simplest way to do this check the accuracy of the predictions. For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate.

NOTE: *You could check the accuracy manually by using* `signnames.csv` *(same directory). This file has a mapping from the class id (0-42) to the corresponding sign name. So, you could take the class id the model outputs, lookup the name in* `signnames.csv` *and see if it matches the sign from the image.*

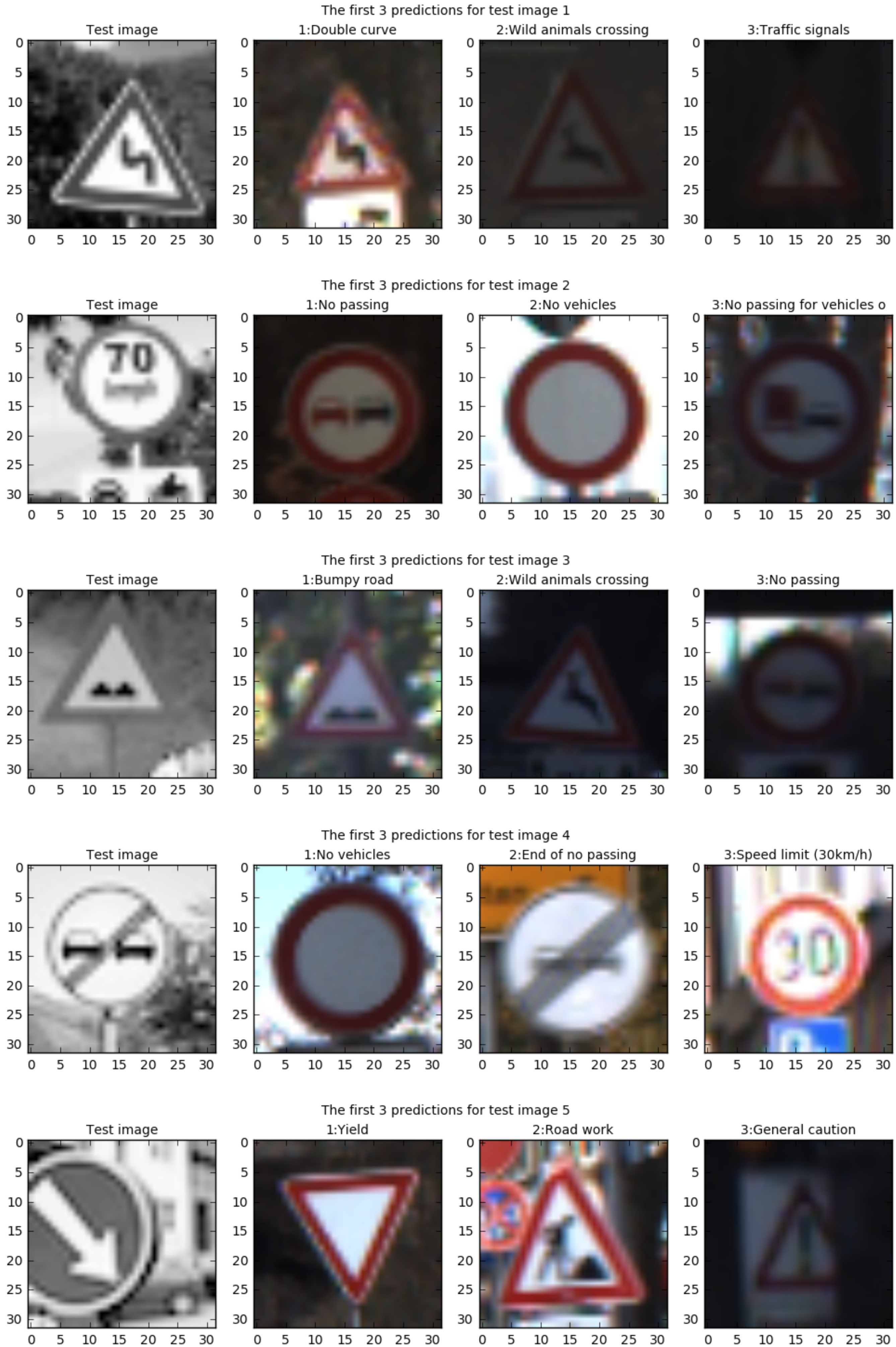
Answer: From the prediction I did to the candidate image, the testing accuracy is 40%, however, the model prediction accuracy on the training set was 97%. As a result, I believe my model did not perform well in the real world situation. One possible reason would be the condidate image is not close to the training image. For example, the second candidate image is 70 km speed limit sign but the number "70" is smaller than usual ones and there is also a text "kmph" below. Another possible reason would be the condidate image is not centered such as the last one.

In [204]:
```
### Visualize the softmax probabilities here.
### Feel free to use as many code cells as needed.
topk = 3
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    TOPKV = sess.run(tf.nn.top_k(logits, k=topk), feed_dict={x: X_gray_real})
print(TOPKV.indices)
print(TOPKV.values)
```

```
[[21 31 26]
 [ 9 15 10]
 [22 31  9]
 [15 41  1]
 [13 25 18]]
[[ 51.00822449  32.92576599  22.51049423]
 [ 19.29883385  18.24463463  18.21645164]
 [ 14.06941795  14.03292561   8.89821815]
 [  3.7897563    3.04293275   1.70141268]
 [ 36.35060883  30.27230263   9.6018858 ]]
```

In [206]:

```
for i in range(n_real):
    fig = plt.figure(figsize=(12, 3))
    plt.subplot(141)
    image = cv2.resize(cv2.imread('data/trafficSign/' + str(i+1) +'.png', 0),
(32,32), interpolation = cv2.INTER_AREA)
    plt.imshow(image, cmap="Greys_r")
    plt.title("Test image",  fontsize=10)
    #plt.imshow(X_gray_real[i,:,:,0], cmap="Greys_r")
    signClass = TOPKV.indices[i, 0]
    signName = SignNames[signClass]
    plt.subplot(142)
    show_example_sign(X_train, y_train, signClass)
    plt.title("1:{}".format(signName[0:min(25,len(signName))]),  fontsize=10)

    signClass = TOPKV.indices[i, 1]
    signName = SignNames[signClass]
    plt.subplot(143)
    show_example_sign(X_train, y_train, signClass)
    plt.title("2:{}".format(signName[0:min(25,len(signName))]),  fontsize=10)

    signClass = TOPKV.indices[i, 2]
    signName = SignNames[signClass]
    plt.subplot(144)
    show_example_sign(X_train, y_train, signClass)
    plt.title("3:{}".format(signName[0:min(25,len(signName))]),  fontsize=10)

    #signClass = TOPKV.indices[i, 3]
    #signName = SignNames[signClass]
    #plt.subplot(165)
    #show_example_sign(X_train, y_train, signClass)
    #plt.title("4:{}".format(signName[0:min(20,len(signName))]),  fontsize=10)

    #signClass = TOPKV.indices[i, 4]
    #signName = SignNames[signClass]
    #plt.subplot(166)
    #show_example_sign(X_train, y_train, signClass)
    #plt.title("5:{}".format(signName[0:min(20,len(signName))]),  fontsize=10)
    plt.suptitle("The first {} predictions for test image {}".format(topk,i+1))
```


The first 3 predictions for test image 1
Test image
1:Double curve
2:Wild animals crossing
3:Traffic signals
The first 3 predictions for test image 2
Test image
1:No passing
2:No vehicles
3:No passing for vehicles o
The first 3 predictions for test image 3
Test image
1:Bumpy road
2:Wild animals crossing
3:No passing
The first 3 predictions for test image 4
Test image
1:No vehicles
2:End of no passing
3:Speed limit (30km/h)
The first 3 predictions for test image 5
Test image
1:Yield
2:Road work
3:General caution

Question 8

*Use the model's softmax probabilities to visualize the **certainty** of its predictions, [`tf.nn.top_k` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k)](https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k) could prove helpful here. Which predictions is the model certain of? Uncertain? If the model was incorrect in its initial prediction, does the correct prediction appear in the top k? (k should be 5 at most)*

`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:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.07893497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

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

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

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

Answer: Only the first prediction is certain and others are not based on the softmax probabilities. For example, the first softmax probability value for the first candidate image is much larger than other two values for it. The first and second softmax probability values for the second candidate image is much larger than the third value for it. All first three softmax probability value are almost equal for the rest three candidate images. In addition, image 4 was incorrect classified but the correct prediction is shown in first 3 prediction.

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.

```
In [ ]:
```