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 [1]:

```
# Load pickled data
import pickle

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

training_file = './traffic-signs-data/train.p'
testing_file = './traffic-signs-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)

FEATURES = 'features'
LABELS = 'labels'
SIZES = 'sizes'
COORDS = 'coords'

X_train_raw, y_train_raw, Sizes_train, Coords_train = train[FEATURES], train[LABELS], train[SIZE
S], train[COORDS]
X_test, y_test, Sizes_test, Coords_test = test[FEATURES], test[LABELS], train[SIZES], train[COOR
DS]

print(X_train_raw.shape,y_train_raw.shape,X_test.shape,y_test.shape)
print(Sizes_train.shape,Sizes_test.shape,Coords_train.shape,Coords_test.shape)
```

```
(39209, 32, 32, 3) (39209,) (12630, 32, 32, 3) (12630,)
(39209, 2) (39209, 2) (39209, 4) (39209, 4)
```

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 [2]:

```
### Replace each question mark with the appropriate value.

# TODO: Number of training examples
n_train_raw = X_train_raw.shape[0]

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

# TODO: What's the shape of an traffic sign image?
image_shape = X_test.shape

# TODO: How many unique classes/labels there are in the dataset.
n_classes = 43

print("Number of training examples =", n_train_raw)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

```
Number of training examples = 39209
Number of testing examples = 12630
Image data shape = (12630, 32, 32, 3)
Number of classes = 43
```

In [3]:

```
#jitter generating function implemented  by Vivek Yadav
import cv2
def transform_image(img,ang_range,shear_range,trans_range):
    '''
    This function transforms images to generate new images.
    The function takes in following arguments,
    1- Image
    2- ang_range: Range of angles for rotation
    3- shear_range: Range of values to apply affine transform to
    4- trans_range: Range of values to apply translations over.

    A Random uniform distribution is used to generate different parameters for transformation

    '''
    # Rotationsave_file
    #print(img.shape)

    ang_rot = np.random.uniform(ang_range)-ang_range/2
    rows,cols,ch = img.shape
    Rot_M = cv2.getRotationMatrix2D((cols/2,rows/2),ang_rot,1)

    # Translation
    tr_x = trans_range*np.random.uniform()-trans_range/2
    tr_y = trans_range*np.random.uniform()-trans_range/2
    Trans_M = np.float32([[1,0,tr_x],[0,1,tr_y]])

    # Shear
    pts1 = np.float32([[5,5],[20,5],[5,20]])

    pt1 = 5+shear_range*np.random.uniform()-shear_range/2
    pt2 = 20+shear_range*np.random.uniform()-shear_range/2

    pts2 = np.float32([[pt1,5],[pt2,pt1],[5,pt2]])

    shear_M = cv2.getAffineTransform(pts1,pts2)

    img = cv2.warpAffine(img,Rot_M,(cols,rows))
    img = cv2.warpAffine(img,Trans_M,(cols,rows))
    img = cv2.warpAffine(img,shear_M,(cols,rows))
    #print(img.shape)
    return img
```

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.

My tried actions;first,get a view of random image ,then,view different label image of the same class id of the traffic sign.

In [4]:

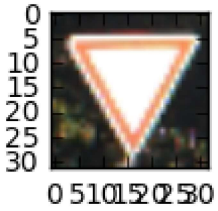
```
### Data exploration visualization goes here.
### Feel free to use as many code cells as needed.
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline

#labelID=33
#while 1:
index = random.randint(0, len(X_train_raw))
#    if y_train[index]==labelID:break

image = X_train_raw[index].squeeze()


plt.figure(figsize=(1,1))
plt.imshow(image)
#plt.imshow(image, cmap="gray")
print("the corresponding label/class id of the traffic sign is ",y_train_raw[index])
```

```
the corresponding label/class id of the traffic sign is  13
```



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 [5]:

```
# see how many input data for each class id of traffic sign
import numpy as np
import random
num_classID=np.zeros(n_classes)

for i in range(len(y_train_raw)):
    num_classID[y_train_raw[i]]+=1
n_label_max=np.max(num_classID)
print(num_classID)
print(n_label_max)
```

```
[  210.  2220.  2250.  1410.  1980.  1860.   420.  1440.  1410.  1470.
  2010.  1320.  2100.  2160.   780.   630.   420.  1110.  1200.   210.
   360.   330.   390.   510.   270.  1500.   600.   240.   540.   270.
   450.   780.   240.   689.   420.  1200.   390.   210.  2070.   300.
   360.   240.   240.]
2250.0
```

In [31]:

```
#split training dato into training and validation set
from sklearn.utils import shuffle
ration_validation=0.3
X_train_raw,y_train_raw = shuffle(X_train_raw, y_train_raw)

n_validation=int(ration_validation*n_train_raw)
n_train=n_train_raw-n_validation

X_train_unjittered= X_train_raw[:n_train]
y_train_unjittered= y_train_raw[:n_train]
X_validation_unjittered = X_train_raw[n_train:]
y_validation_unjittered = y_train_raw[n_train:]
print(n_train,n_validation)
print(X_train_unjittered.shape,y_train_unjittered.shape,X_validation_unjittered.shape,y_validati
on_unjittered.shape)

num_train_classID=np.zeros(n_classes)
num_validation_classID=np.zeros(n_classes)

for i in range(n_train):
    num_train_classID[y_train_unjittered[i]]+=1
n_train_label_max=np.max(num_train_classID)
print(num_train_classID)
print("the maximum number of a label in training set is ",n_train_label_max)

for i in range(n_validation):
    num_validation_classID[y_validation_unjittered[i]]+=1
n_validation_label_max=np.max(num_validation_classID)
print(num_validation_classID)
print("the maximum number of a label in validation set is ",n_validation_label_max)
```

```
27447 11762
(27447, 32, 32, 3) (27447,) (11762, 32, 32, 3) (11762,)
[  154.  1545.  1584.   992.  1421.  1320.   297.  1027.   979.  1051.
  1392.   918.  1458.  1471.   526.   448.   296.   740.   857.   141.
   258.   234.   268.   350.   191.  1048.   449.   167.   375.   194.
   314.   525.   173.   490.   292.   861.   267.   127.  1436.   216.
   251.   168.   176.]
the maximum number of a label in training set is  1584.0
[  56.  675.  666.  418.  559.  540.  123.  413.  431.  419.  618.  402.
  642.  689.  254.  182.  124.  370.  343.   69.  102.   96.  122.  160.
   79.  452.  151.   73.  165.   76.  136.  255.   67.  199.  128.  339.
  123.   83.  634.   84.  109.   72.   64.]
the maximum number of a label in validation set is  689.0
```

In [32]:

```
# generate jitter data ,making data balanceed
X_train=[]
y_train=[]
X_validation=[]
y_validation=[]
n_times=1
print("n_train" ,n_train)
for m in range(n_times):
    for i in range(n_train):
         for j in range(n_classes):
            if y_train_unjittered[i]==j:
                for k in range(int(round(n_train_label_max/num_train_classID[j]))):
                    img = transform_image(X_train_unjittered[i],10,5,5)
                    X_train.append(img)
                    y_train.append(y_train_unjittered[i])
print("the jitter trainning data size is",len(X_train))
for i in range(n_train):
    X_train.append(X_train_unjittered[i])
    y_train.append(y_train_unjittered[i])

X_train=np.array(X_train)
y_train=np.array(y_train)
n_train=len(X_train)
print("the shape of final X_train ,y_train is",X_train.shape,y_train.shape)


for m in range(n_times):
    for i in range(n_validation):
        for j in range(n_classes):
            if y_validation_unjittered[i]==j:
                for k in range(int(round(n_validation_label_max/num_validation_classID[j]))):
                    img = transform_image(X_validation_unjittered[i],10,5,5)
                    X_validation.append(img)
                    y_validation.append(y_validation_unjittered[i])

print("the jitter trainning data size is",len(X_validation))
for i in range(n_validation):
    X_validation.append(X_validation_unjittered[i])
    y_validation.append(y_validation_unjittered[i])

X_validation=np.array(X_validation)
y_validation=np.array(y_validation)
n_validation=len(X_validation)


print("the shape of final X_validation ,y_validation is",X_validation.shape,y_validation.shape)

X_train,y_train = shuffle(X_train, y_train)
X_validation,y_validation = shuffle(X_validation, y_validation)

"""
n_validation=n_train_raw

X_validation=X_train_raw
y_validation=y_train_raw

X_train=[]
y_train=[]
#for k in range(4):
for i in range(len(y_train_raw)):
    for j in range(n_classes):
```

```
        if y_train_raw[i]==j:
            for k in range(int(round(n_label_max/num_classID[j]))):
                img = transform_image(image,10,5,2)
                X_train.append(img)
                y_train.append(y_train_raw[i])
X_train=np.array(X_train)
y_train=np.array(y_train)
n_train=len(X_train)

print("the shape of X_train y_train is",X_train.shape,y_train.shape)


"""



"""
image =imread()
plt.imshow(image);
plt.axis('off');
plt.show()
gs1 = gridspec.GridSpec(10, 10)
gs1.update(wspace=0.01, hspace=0.02) # set the spacing between axes.
plt.figure(figsize=(12,12))
for i in range(100):
    ax1 = plt.subplot(gs1[i])
    ax1.set_xticklabels([])
    ax1.set_yticklabels([])
    ax1.set_aspect('equal')
    img = transform_image(image,20,10,5)

    plt.subplot(10,10,i+1)
    plt.imshow(img)
    plt.axis('off')

plt.show()
"""
```

```
n_train 27447
the jitter trainning data size is 69294
the shape of final X_train ,y_train is (96741, 32, 32, 3) (96741,)
the jitter trainning data size is 30260
the shape of final X_validation ,y_validation is (42022, 32, 32, 3) (42022,)
```

Out[32]:

```
"\nimage =imread()\nplt.imshow(image);\nplt.axis('off');\nplt.show()\ngs1 = gridsp
ec.GridSpec(10, 10)\ngs1.update(wspace=0.01, hspace=0.02) # set the spacing betwee
n axes. \nplt.figure(figsize=(12,12))\nfor i in range(100):\n    ax1 = plt.subplot
(gs1[i])\n    ax1.set_xticklabels([])\n    ax1.set_yticklabels([])\n    ax1.set_as
pect('equal')\n    img = transform_image(image,20,10,5)\n\n    plt.subplot(10,10,i
+1)\n    plt.imshow(img)\n    plt.axis('off')\n\nplt.show()\n"
```

In [33]:

```
#preprocessing the input data  :RGB to YUV  Gray Y = R*0.299 + G*0.587 + B*0.114
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

#X_train_gray=np.array([n_train,32,32,1])
#X_test_gray=np.array([n_test,32,32,1])

X_train_gray=np.array(rgb2gray(X_train),dtype=np.float32).reshape((n_train,32,32,1))
X_test_gray=np.array(rgb2gray(X_test),dtype=np.float32).reshape((n_test,32,32,1))
X_validation_gray=np.array(rgb2gray(X_validation),dtype=np.float32).reshape((n_validation,32,32,1


print("the shape of X_train_gray is ",X_train_gray.shape)
print("the shape of X_validation_gray is ",X_validation_gray.shape)
print("the shape of X_test_gray is ",X_test_gray.shape)


#print("y_test",y_test)
```

```
the shape of X_train_gray is  (96741, 32, 32, 1)
the shape of X_validation_gray is  (42022, 32, 32, 1)
the shape of X_test_gray is  (12630, 32, 32, 1)
```

In [34]:

```
#data normalizing
import tensorflow as tf

def image_normalizing(data):
    d=np.array(data)
    shape=d.shape
    d=d.reshape(d.size)
    for i in range(d.size):
        d[i]=np.float32((d[i]-128)/128)
    return d.reshape(shape)

X_train_gray_normal=image_normalizing(X_train_gray)
X_test_gray_normal=image_normalizing(X_test_gray)
X_validation_gray_normal=image_normalizing(X_validation_gray)

print("the shape of X_train_gray_normal is ",X_train_gray_normal.shape)
print("the shape of X_validation_gray_normal is ",X_validation_gray_normal.shape)
print("the shape of X_test_gray_normal is ",X_test_gray_normal.shape)


#print("X_train_gray_normal",X_train_gray_normal)
#print("one_hot_y_test",one_hot_y_test)
```

In [39]:

```
# using the leNet structure fist


from tensorflow.contrib.layers import flatten
#shuffling the data pair
X_train_gray_normal,y_train = shuffle(X_train_gray_normal, y_train)

EPOCHS = 200
BATCH_SIZE = 128
#dropout_rate = 0.750

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

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

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

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

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

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

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

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

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(84))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

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

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output =  n_classes,in this case equals 43
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(84,  n_classes), mean = mu, stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros( n_classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b
```

```
    return logits


def LeNetModify1(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x16.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 16), mean = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(16))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

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

    # SOLUTION: Pooling. Input = 28x28x16. Output = 14x14x16.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x32.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 32), mean = mu, stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(32))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

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

    # SOLUTION: Pooling. Input = 10x10x32. Output = 5x5x32.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x32. Output = 800.
    fc0   = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 800. Output = 256.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(800, 256), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(256))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    # SOLUTION: Layer 4: Fully Connected. Input = 256. Output = 128.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(256, 128), mean = mu, stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(128))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

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

    # SOLUTION: Layer 5: Fully Connected. Input = 128. Output =  n_classes,in this case equals 4
3
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(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 [40]:

```
# training pipeline

x = tf.placeholder(tf.float32, (None, 32, 32,1))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)

rate = 0.001
#logits=LeNet(x)
logits = LeNetModify1(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)
```

In [41]:

```
#evaluating function

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))

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 [42]:

```
# model training
save_file='./model.ckpt'

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    num_examples = len(X_train_gray_normal)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train_gray_normal, y_train = shuffle(X_train_gray_normal, y_train)
        #print("X_train_gray_normal.shape",X_train_gray_normal.shape)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train_gray_normal[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})

        validation_accuracy = evaluate(X_validation_gray_normal, y_validation)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

    # try test dat accuray
    test_accuracy = evaluate(X_test_gray_normal, y_test)
    print("Test Accuracy = {:.3f}".format(test_accuracy))

    try:
        saver
    except NameError:
        saver = tf.train.Saver()
    print(saver.save(sess,save_file))
    print("Model saved")
```

```
WARNING:tensorflow:From <ipython-input-42-2a6bacceaf89>:5 in <module>.: initialize
_all_variables (from tensorflow.python.ops.variables) is deprecated and will be re
moved after 2017-03-02.
Instructions for updating:
Use `tf.global_variables_initializer` instead.
Training...

EPOCH 1 ...
Validation Accuracy = 0.840

EPOCH 2 ...
Validation Accuracy = 0.906

EPOCH 3 ...
Validation Accuracy = 0.941

EPOCH 4 ...
Validation Accuracy = 0.952

EPOCH 5 ...
Validation Accuracy = 0.952

EPOCH 6 ...
Validation Accuracy = 0.954

EPOCH 7 ...
Validation Accuracy = 0.954

EPOCH 8 ...
Validation Accuracy = 0.961

EPOCH 9 ...
Validation Accuracy = 0.963

EPOCH 10 ...
Validation Accuracy = 0.961

EPOCH 11 ...
Validation Accuracy = 0.963

EPOCH 12 ...
Validation Accuracy = 0.965

EPOCH 13 ...
Validation Accuracy = 0.965

EPOCH 14 ...
Validation Accuracy = 0.967

EPOCH 15 ...
Validation Accuracy = 0.968

EPOCH 16 ...
Validation Accuracy = 0.970

EPOCH 17 ...
Validation Accuracy = 0.965

EPOCH 18 ...
Validation Accuracy = 0.972
```

```
EPOCH 19 ...
Validation Accuracy = 0.970

EPOCH 20 ...
Validation Accuracy = 0.969

EPOCH 21 ...
Validation Accuracy = 0.973

EPOCH 22 ...
Validation Accuracy = 0.969

EPOCH 23 ...
Validation Accuracy = 0.979

EPOCH 24 ...
Validation Accuracy = 0.973

EPOCH 25 ...
Validation Accuracy = 0.975

EPOCH 26 ...
Validation Accuracy = 0.967

EPOCH 27 ...
Validation Accuracy = 0.974

EPOCH 28 ...
Validation Accuracy = 0.974

EPOCH 29 ...
Validation Accuracy = 0.978

EPOCH 30 ...
Validation Accuracy = 0.969

EPOCH 31 ...
Validation Accuracy = 0.979

EPOCH 32 ...
Validation Accuracy = 0.970

EPOCH 33 ...
Validation Accuracy = 0.977

EPOCH 34 ...
Validation Accuracy = 0.975

EPOCH 35 ...
Validation Accuracy = 0.975

EPOCH 36 ...
Validation Accuracy = 0.977

EPOCH 37 ...
Validation Accuracy = 0.974

EPOCH 38 ...
Validation Accuracy = 0.975

EPOCH 39 ...
```

```
Validation Accuracy = 0.979

EPOCH 40 ...
Validation Accuracy = 0.978

EPOCH 41 ...
Validation Accuracy = 0.978

EPOCH 42 ...
Validation Accuracy = 0.975

EPOCH 43 ...
Validation Accuracy = 0.978

EPOCH 44 ...
Validation Accuracy = 0.977

EPOCH 45 ...
Validation Accuracy = 0.978

EPOCH 46 ...
Validation Accuracy = 0.977

EPOCH 47 ...
Validation Accuracy = 0.977

EPOCH 48 ...
Validation Accuracy = 0.976

EPOCH 49 ...
Validation Accuracy = 0.978

EPOCH 50 ...
Validation Accuracy = 0.978

EPOCH 51 ...
Validation Accuracy = 0.976

EPOCH 52 ...
Validation Accuracy = 0.966

EPOCH 53 ...
Validation Accuracy = 0.977

EPOCH 54 ...
Validation Accuracy = 0.977

EPOCH 55 ...
Validation Accuracy = 0.976

EPOCH 56 ...
Validation Accuracy = 0.981

EPOCH 57 ...
Validation Accuracy = 0.974

EPOCH 58 ...
Validation Accuracy = 0.977

EPOCH 59 ...
Validation Accuracy = 0.975
```

```
EPOCH 60 ...
Validation Accuracy = 0.979

EPOCH 61 ...
Validation Accuracy = 0.977

EPOCH 62 ...
Validation Accuracy = 0.979

EPOCH 63 ...
Validation Accuracy = 0.979

EPOCH 64 ...
Validation Accuracy = 0.979

EPOCH 65 ...
Validation Accuracy = 0.976

EPOCH 66 ...
Validation Accuracy = 0.983

EPOCH 67 ...
Validation Accuracy = 0.979

EPOCH 68 ...
Validation Accuracy = 0.981

EPOCH 69 ...
Validation Accuracy = 0.978

EPOCH 70 ...
Validation Accuracy = 0.981

EPOCH 71 ...
Validation Accuracy = 0.980

EPOCH 72 ...
Validation Accuracy = 0.977

EPOCH 73 ...
Validation Accuracy = 0.979

EPOCH 74 ...
Validation Accuracy = 0.979

EPOCH 75 ...
Validation Accuracy = 0.970

EPOCH 76 ...
Validation Accuracy = 0.980

EPOCH 77 ...
Validation Accuracy = 0.980

EPOCH 78 ...
Validation Accuracy = 0.978

EPOCH 79 ...
Validation Accuracy = 0.979
```

```
EPOCH 80 ...
Validation Accuracy = 0.982

EPOCH 81 ...
Validation Accuracy = 0.980

EPOCH 82 ...
Validation Accuracy = 0.978

EPOCH 83 ...
Validation Accuracy = 0.975

EPOCH 84 ...
Validation Accuracy = 0.978

EPOCH 85 ...
Validation Accuracy = 0.977

EPOCH 86 ...
Validation Accuracy = 0.979

EPOCH 87 ...
Validation Accuracy = 0.980

EPOCH 88 ...
Validation Accuracy = 0.977

EPOCH 89 ...
Validation Accuracy = 0.980

EPOCH 90 ...
Validation Accuracy = 0.980

EPOCH 91 ...
Validation Accuracy = 0.979

EPOCH 92 ...
Validation Accuracy = 0.980

EPOCH 93 ...
Validation Accuracy = 0.975

EPOCH 94 ...
Validation Accuracy = 0.981

EPOCH 95 ...
Validation Accuracy = 0.982

EPOCH 96 ...
Validation Accuracy = 0.982

EPOCH 97 ...
Validation Accuracy = 0.976

EPOCH 98 ...
Validation Accuracy = 0.983

EPOCH 99 ...
Validation Accuracy = 0.981

EPOCH 100 ...
```

```
Validation Accuracy = 0.979

EPOCH 101 ...
Validation Accuracy = 0.977

EPOCH 102 ...
Validation Accuracy = 0.975

EPOCH 103 ...
Validation Accuracy = 0.982

EPOCH 104 ...
Validation Accuracy = 0.980

EPOCH 105 ...
Validation Accuracy = 0.980

EPOCH 106 ...
Validation Accuracy = 0.979

EPOCH 107 ...
Validation Accuracy = 0.981

EPOCH 108 ...
Validation Accuracy = 0.978

EPOCH 109 ...
Validation Accuracy = 0.982

EPOCH 110 ...
Validation Accuracy = 0.980

EPOCH 111 ...
Validation Accuracy = 0.980

EPOCH 112 ...
Validation Accuracy = 0.980

EPOCH 113 ...
Validation Accuracy = 0.981

EPOCH 114 ...
Validation Accuracy = 0.980

EPOCH 115 ...
Validation Accuracy = 0.977

EPOCH 116 ...
Validation Accuracy = 0.978

EPOCH 117 ...
Validation Accuracy = 0.979

EPOCH 118 ...
Validation Accuracy = 0.981

EPOCH 119 ...
Validation Accuracy = 0.983

EPOCH 120 ...
Validation Accuracy = 0.979
```

```
EPOCH 121 ...
Validation Accuracy = 0.977

EPOCH 122 ...
Validation Accuracy = 0.982

EPOCH 123 ...
Validation Accuracy = 0.979

EPOCH 124 ...
Validation Accuracy = 0.982

EPOCH 125 ...
Validation Accuracy = 0.981

EPOCH 126 ...
Validation Accuracy = 0.975

EPOCH 127 ...
Validation Accuracy = 0.982

EPOCH 128 ...
Validation Accuracy = 0.980

EPOCH 129 ...
Validation Accuracy = 0.984

EPOCH 130 ...
Validation Accuracy = 0.979

EPOCH 131 ...
Validation Accuracy = 0.983

EPOCH 132 ...
Validation Accuracy = 0.983

EPOCH 133 ...
Validation Accuracy = 0.980

EPOCH 134 ...
Validation Accuracy = 0.980

EPOCH 135 ...
Validation Accuracy = 0.978

EPOCH 136 ...
Validation Accuracy = 0.974

EPOCH 137 ...
Validation Accuracy = 0.981

EPOCH 138 ...
Validation Accuracy = 0.977

EPOCH 139 ...
Validation Accuracy = 0.981

EPOCH 140 ...
Validation Accuracy = 0.985
```

```
EPOCH 141 ...
Validation Accuracy = 0.977

EPOCH 142 ...
Validation Accuracy = 0.982

EPOCH 143 ...
Validation Accuracy = 0.981

EPOCH 144 ...
Validation Accuracy = 0.984

EPOCH 145 ...
Validation Accuracy = 0.981

EPOCH 146 ...
Validation Accuracy = 0.983

EPOCH 147 ...
Validation Accuracy = 0.978

EPOCH 148 ...
Validation Accuracy = 0.980

EPOCH 149 ...
Validation Accuracy = 0.984

EPOCH 150 ...
Validation Accuracy = 0.983

EPOCH 151 ...
Validation Accuracy = 0.979

EPOCH 152 ...
Validation Accuracy = 0.980

EPOCH 153 ...
Validation Accuracy = 0.977

EPOCH 154 ...
Validation Accuracy = 0.984

EPOCH 155 ...
Validation Accuracy = 0.983

EPOCH 156 ...
Validation Accuracy = 0.976

EPOCH 157 ...
Validation Accuracy = 0.984

EPOCH 158 ...
Validation Accuracy = 0.983

EPOCH 159 ...
Validation Accuracy = 0.981

EPOCH 160 ...
Validation Accuracy = 0.984

EPOCH 161 ...
```

```
Validation Accuracy = 0.981

EPOCH 162 ...
Validation Accuracy = 0.982

EPOCH 163 ...
Validation Accuracy = 0.981

EPOCH 164 ...
Validation Accuracy = 0.982

EPOCH 165 ...
Validation Accuracy = 0.978

EPOCH 166 ...
Validation Accuracy = 0.985

EPOCH 167 ...
Validation Accuracy = 0.979

EPOCH 168 ...
Validation Accuracy = 0.979

EPOCH 169 ...
Validation Accuracy = 0.979

EPOCH 170 ...
Validation Accuracy = 0.977

EPOCH 171 ...
Validation Accuracy = 0.983

EPOCH 172 ...
Validation Accuracy = 0.981

EPOCH 173 ...
Validation Accuracy = 0.982

EPOCH 174 ...
Validation Accuracy = 0.981

EPOCH 175 ...
Validation Accuracy = 0.982

EPOCH 176 ...
Validation Accuracy = 0.982

EPOCH 177 ...
Validation Accuracy = 0.979

EPOCH 178 ...
Validation Accuracy = 0.983

EPOCH 179 ...
Validation Accuracy = 0.982

EPOCH 180 ...
Validation Accuracy = 0.983

EPOCH 181 ...
Validation Accuracy = 0.981
```

```
EPOCH 182 ...
Validation Accuracy = 0.984

EPOCH 183 ...
Validation Accuracy = 0.983

EPOCH 184 ...
Validation Accuracy = 0.978

EPOCH 185 ...
Validation Accuracy = 0.979

EPOCH 186 ...
Validation Accuracy = 0.984

EPOCH 187 ...
Validation Accuracy = 0.982

EPOCH 188 ...
Validation Accuracy = 0.982

EPOCH 189 ...
Validation Accuracy = 0.982

EPOCH 190 ...
Validation Accuracy = 0.981

EPOCH 191 ...
Validation Accuracy = 0.981

EPOCH 192 ...
Validation Accuracy = 0.984

EPOCH 193 ...
Validation Accuracy = 0.983

EPOCH 194 ...
Validation Accuracy = 0.978

EPOCH 195 ...
Validation Accuracy = 0.981

EPOCH 196 ...
Validation Accuracy = 0.982

EPOCH 197 ...
Validation Accuracy = 0.983

EPOCH 198 ...
Validation Accuracy = 0.984

EPOCH 199 ...
Validation Accuracy = 0.982

EPOCH 200 ...
Validation Accuracy = 0.985

Test Accuracy = 0.955
./model.ckpt
Model saved
```

In [8]:

```
# evalute the model
import tensorflow as tf

with tf.Session() as sess:
    #saver = tf.train.Saver()
    #saver.restore(sess,save_file)
    save_file='./model.ckpt'
    ckpt = tf.train.get_checkpoint_state(save_file)
    if ckpt and ckpt.model_checkpoint_path:
        print("Continue training from the model {}".format(ckpt.model_checkpoint_path))
        #print(saver)
        #print(saver.restore(sess, ckpt.model_checkpoint_path))
    ckpt = tf.train.import_meta_graph('model.ckpt.meta')
    ckpt.restore(sess, './model.ckpt')


    #saver.restore(sess,save_file)
    #loader.restore(sess, tf.train.latest_checkpoint('./'))

    test_accuracy = evaluate(X_test_gray_normal, y_test)
    #print(test_accuracy.eval(session=sess))
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-477002a7cb18> in <module>()
      2 import tensorflow as tf
      3
----> 4 correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y,
1))
      5 accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))
      6

NameError: name 'logits' is not defined
```

Question 1

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

Answer:

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.

first ,i split the data into trainding data/validation data with a ratio of 27447/ 11762, with same model, it can reach 94% accurate rate at maximum. second, i seperately generate the jittered training data and valdation data for previous splitted trainding data and validation data with a ratio of 96741/30260 ,then the accurate rise up besides,for both raw data and jittered data ,i convert all the data from rgb to y channel of yuv, and then normalizing them

In [ ]:

```
#split training dato into training and validation set
from sklearn.utils import shuffle

X_train_raw,y_train_raw = shuffle(X_train_raw, y_train_raw)

n_validation=int(0.3*n_train_raw)
n_train=n_train_raw-n_validation

X_train_unjittered= X_train_raw[:n_train]
y_train_unjittered= y_train_raw[:n_train]
X_validation_unjittered = X_train_raw[n_train:]
y_validation_unjittered = y_train_raw[n_train:]
```

In [ ]:

```
# generate jitter data ,making data balanceed
X_train=[]
y_train=[]
X_validation=[]
y_validation=[]
n_times=1
print("n_train" ,n_train)
for m in range(n_times):
    for i in range(n_train):
         for j in range(n_classes):
            if y_train_unjittered[i]==j:
                for k in range(int(round(n_train_label_max/num_train_classID[j]))):
                    img = transform_image(X_train_unjittered[i],10,5,5)
                    X_train.append(img)
                    y_train.append(y_train_unjittered[i])
print("the jitter trainning data size is",len(X_train))
for i in range(n_train):
    X_train.append(X_train_unjittered[i])
    y_train.append(y_train_unjittered[i])

X_train=np.array(X_train)
y_train=np.array(y_train)
n_train=len(X_train)
print("the shape of final X_train ,y_train is",X_train.shape,y_train.shape)


for m in range(n_times):
    for i in range(n_validation):
        for j in range(n_classes):
            if y_validation_unjittered[i]==j:
                for k in range(int(round(n_validation_label_max/num_validation_classID[j]))):
                    img = transform_image(X_validation_unjittered[i],10,5,5)
                    X_validation.append(img)
                    y_validation.append(y_validation_unjittered[i])

print("the jitter trainning data size is",len(X_validation))
for i in range(n_validation):
    X_validation.append(X_validation_unjittered[i])
    y_validation.append(y_validation_unjittered[i])

X_validation=np.array(X_validation)
y_validation=np.array(y_validation)
n_validation=len(X_validation)


print("the shape of final X_validation ,y_validation is",X_validation.shape,y_validation.shape)

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

In [ ]:

```
#preprocessing the input data  :RGB to YUV  Gray Y = R*0.299 + G*0.587 + B*0.114
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

#X_train_gray=np.array([n_train,32,32,1])
#X_test_gray=np.array([n_test,32,32,1])

X_train_gray=np.array(rgb2gray(X_train),dtype=np.float32).reshape((n_train,32,32,1))
X_test_gray=np.array(rgb2gray(X_test),dtype=np.float32).reshape((n_test,32,32,1))
X_validation_gray=np.array(rgb2gray(X_validation),dtype=np.float32).reshape((n_validation,32,32,1
```

In [ ]:

```
#data normalizing
import tensorflow as tf

def image_normalizing(data):
    d=np.array(data)
    shape=d.shape
    d=d.reshape(d.size)
    for i in range(d.size):
        d[i]=np.float32((d[i]-128)/128)
    return d.reshape(shape)

X_train_gray_normal=image_normalizing(X_train_gray)
X_test_gray_normal=image_normalizing(X_test_gray)
X_validation_gray_normal=image_normalizing(X_validation_gray)
```

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:

Define your architecture here.

Feel free to use as many code cells as needed.

i keep the testing data untouched ,just the same as the input test data. and seperate the original training data into two part ,the training data with jittered traing data, the validation data with jittered validation data, besides with jittered data ,i balance the data samples for each traffic sign label avoiding from the tendency model by the unbalanced data imput. in order to make the model more robust, i refer to thejitter generating function implemented by Vivek Yadav which can ramdomly rotate,shear,translation transform the input data

with the generated data ,the training data become larger and more random jittered whichi tend to output a more robutst model.

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) from the classroom.

Answer:

the model final.

Feel free to use as many code cells as needed.

first,i use a simple linear regression structure whichi yield not good accurate. then i choose a lenet model ,it turns better. and finaly i choose a modified lenet model,whichi shows a satisfied result.

the architeture is below: first: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x16. with a 16 channels different 5x5x1 filters followed by a relu activation then resampling by a max pooling with a [1,2,2,1]stride which output 14x14x16. secondly,Layer 2: Convolutional. Output = 10x10x32. with a 32 channels different 5x5x1 filters follows by another relu activation and then another max pooling with a [1,2,2,1]stride which output 5x5x32.. thirdly ,the flatten layer with an output of 800 nodes fouthly,fully Connected layer of linear regression. with Input = 800. Output = 256. followed by another relu activation fifthly,Fully Connected layer. Input = 256. Output = 128. followed by another relu activation finally,Fully Connected output layer. Input = 128. Output = 43

In [ ]:

```
def LeNetModify1(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x16.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 16), mean = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(16))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

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

    # SOLUTION: Pooling. Input = 28x28x16. Output = 14x14x16.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x32.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 32), mean = mu, stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(32))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

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

    # SOLUTION: Pooling. Input = 10x10x32. Output = 5x5x32.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

    # SOLUTION: Flatten. Input = 5x5x32. Output = 800.
    fc0   = flatten(conv2)

    # SOLUTION: Layer 3: Fully Connected. Input = 800. Output = 256.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(800, 256), mean = mu, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(256))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    # SOLUTION: Layer 4: Fully Connected. Input = 256. Output = 128.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(256, 128), mean = mu, stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(128))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

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

    # SOLUTION: Layer 5: Fully Connected. Input = 128. Output =  n_classes,in this case equals 4
3
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(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
```

Question 4

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

Answer:

In [ ]:

```
i use the  tf.train.AdamOptimizer wih the hyperparameters as belows:
    learning rate = 0.001
    batch size=128
    epochs=200
    ration_validation=0.3
    transform_image(image,ang_range=10,shear_range=5,trans_range=5)
```

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:

In [ ]:

```
first,,i use a simple linear regression structure whichi yield not good accurate.
then i choose a lenet model because i think that the conv net is effective in processing image i
nputs.and i got an accuracy about 89%
then i try a wider modified lenet model,whichi shows a satisfied result of 94%
and i come back to add more generating data to see how much it can improve the accuracy,
and i found it takes more computing time,first i set epoches to 30 ,which i can see the model is
 not converged in the last epoque,
so i set epoches to 200 ,which i can see the model is converged.
and i try to update the batch size to 256 to make it run faster, but it cosumes more memory whic
h my computer cann't afford.
so i finally choose  the batch size 128'
```

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 [ ]:

```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
#np.array(newdata,dtype=np.float32)
```

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:

In [ ]:

```
### Run the predictions here.
### Feel free to use as many code cells as needed.
```

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:

In [ ]:

```
### Visualize the softmax probabilities here.
### Feel free to use as many code cells as needed.
```

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) 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:

> **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 [ ]: