Self-Driving Car Engineer Nanodegree

Deep Learning

Project: Build a Traffic Sign Recognition Classifier

In this notebook, a template is provided for you to implement your functionality in stages, which is required to successfully complete this project. If additional code is required that cannot be included in the notebook, be sure that the Python code is successfully imported and included in your submission if necessary.

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the iPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there is a writeup to complete. The writeup should be completed in a separate file, which can be either a markdown file or a pdf document. There is a write up template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points (https://review.udacity.com/#!/rubrics/481/view) for this project.

The rubric (https://review.udacity.com/#!/rubrics/481/view) contains "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. The stand out suggestions are optional. If you decide to pursue the "stand out suggestions", you can include the code in this Ipython notebook and also discuss the results in the writeup file.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. In addition, Markdown cells can be edited by typically double-clicking the cell to enter edit mode.

Step 0: Load The Data

In [1]:
```python
import pickle

training_file = 'train.p'
validation_file = 'valid.p'
testing_file = 'test.p'

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

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing tuples, (width, height) representing the the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the results. For example, the [pandas shape method (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html)](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html) might be useful for calculating some of the summary results.

Provide a Basic Summary of the Data Set Using Python, Numpy and/or Pandas

In [2]:
```
import numpy as np
import pandas as pd

n_train = len(X_train)
n_test = len(X_test)
image_shape = X_train[0].shape
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)
```

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

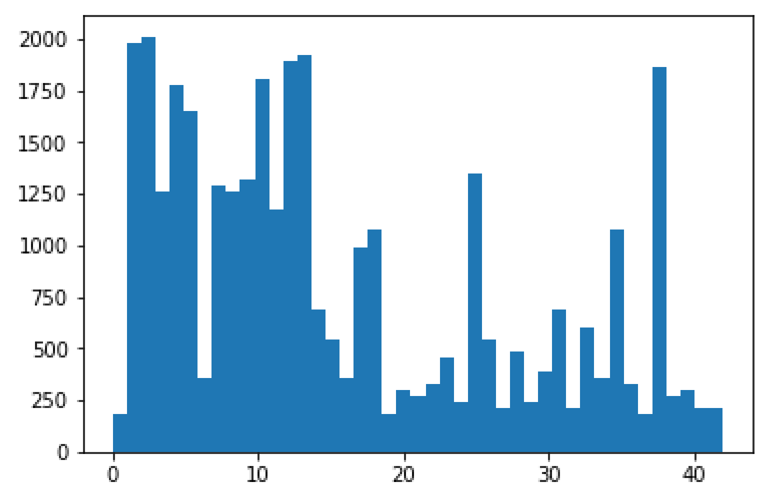
Include an exploratory visualization of the dataset

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

The Matplotlib (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections.

In [3]:
```
import matplotlib.pyplot as plt

%matplotlib inline

plt.hist(y_train, bins=n_classes)
plt.show()
```



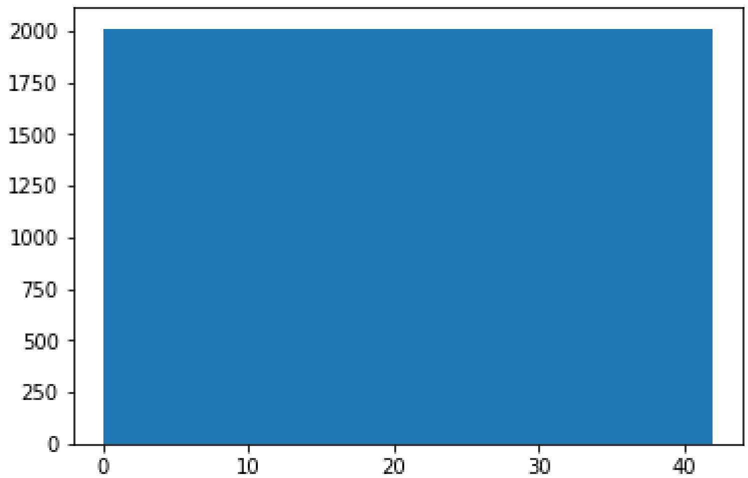
In [4]:
```
import numpy as np

histogram = np.histogram(y_train, n_classes)
```

In [5]:

```
import cv2
import math

# 'Hyperparameter'
ANGLE = 30

# Based on http://stackoverflow.com/a/19110462, need a better way to cal
c transformation matrix...
def warp(img, rotx=0, roty=0, rotz=0, f=2):
    h, w, c = img.shape

    cx = math.cos(rotx)
    sx = math.sin(rotx)
    cy = math.cos(roty)
    sy = math.sin(roty)
    cz = math.cos(rotz)
    sz = math.sin(rotz)

    roto = [
        [cz * cy, cz * sy * sx - sz * cx],
        [sz * cy, sz * sy * sx + cz * cx],
        [-sy, cy * sx]
    ]

    pt = [
        [-w / 2, -h / 2],
        [w / 2, -h / 2],
        [w / 2, h / 2],
        [-w / 2, h / 2]
    ]

    ptt = np.zeros((4, 2), dtype=float)

    for i in range(4):
        pz = pt[i][0] * roto[2][0] + pt[i][1] * roto[2][1]
        ptt[i][0] = w / 2 + (pt[i][0] * roto[0][0] + pt[i][1] * roto[0]
[1]) * f * h / (f * h + pz)
        ptt[i][1] = h / 2 + (pt[i][0] * roto[1][0] + pt[i][1] * roto[1]
[1]) * f * h / (f * h + pz)

    src = np.float32([
        [0, 0],
        [w, 0],
        [w, h],
        [0, h]
    ])

    dst = np.float32([
        [ptt[0][0], ptt[0][1]],
        [ptt[1][0], ptt[1][1]],
        [ptt[2][0], ptt[2][1]],
        [ptt[3][0], ptt[3][1]]
```

```
    ])

    return cv2.warpPerspective(img, cv2.getPerspectiveTransform(src,
dst), (w, h), borderMode=cv2.BORDER_REPLICATE)


def generate_signs_variations(signs, count):
    result = []
    for sign in signs:
        multiple = min(int(math.ceil(count / len(signs))), count - len(r
esult))
        combinations = np.random.randint(-ANGLE, ANGLE, (multiple, 3)) *
 math.pi / 180
        for rx, ry, rz in combinations:
            warped = warp(sign, rx, ry, rz)
            result.append(warped)
    return result


def augment(X_train, y_train):
    n_classes = np.unique(y_train).size
    hist, bins = np.histogram(y_train, bins=n_classes)

    X_aug = []
    y_aug = []

    for i, original_count in zip(np.arange(hist.size + 1), hist):
        multiplier = hist.max() / original_count
        variation_count = int((multiplier - 1) * original_count)
        x_idx, *rest = np.where(y_train == i)
        signs = np.asarray([X_train[j] for j in x_idx], dtype=np.uint8)
        variations = generate_signs_variations(signs, variation_count)
        X_aug += variations
        y_aug += np.full(len(variations), i, dtype=np.uint8).tolist()

    return (np.array(X_aug), np.array(y_aug))
```

In [6]:
```
X_aug, y_aug = augment(X_train, y_train)

X_train = np.concatenate((X_train, X_aug))
y_train = np.concatenate((y_train, y_aug))

plt.hist(y_train, bins=n_classes)
plt.show()
```



Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the German Traffic Sign Dataset (http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

There are various aspects to consider when thinking about this problem:

- Neural network architecture
- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

Here is an example of a published baseline model on this problem (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

NOTE: The LeNet-5 implementation shown in the classroom (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) at the end of the CNN lesson is a solid starting point. You'll have to change the number of classes and possibly the preprocessing, but aside from that it's plug and play!

Pre-process the Data Set (normalization, grayscale, etc.)

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project.

In [7]:
```
from sklearn.utils import shuffle
import cv2

X_train, y_train = shuffle(X_train, y_train)

def gray_normalized(image):
    width, height, channels = image.shape
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.equalizeHist(image)
    image = np.reshape(image, (width, height, 1))
    return image

X_train = np.asarray(list(map(lambda x: gray_normalized(x), X_train)))
X_valid = np.asarray(list(map(lambda x: gray_normalized(x), X_valid)))
X_test = np.asarray(list(map(lambda x: gray_normalized(x), X_test)))
```

Model Architecture

In [8]:
```
from tensorflow.contrib.layers import flatten

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

    # Layer 1: 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='VA
LID') + conv1_b

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

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

    # Layer 2: Convolutional. Input = 14x14x6. 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

    # Activation.
    conv2 = tf.nn.relu(conv2)
```

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

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

    # Layer 3: Fully Connected. Input = 400. Output = 190.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(400, 190), mean = mu,
 stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(190))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    # Layer 4: Fully Connected. Input = 190. Output = 90.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(190, 90), mean = mu,
 stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(90))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

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

    # Layer 5: Fully Connected. Input = 90. Output = 43.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(90, 43), mean = mu,
stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros(43))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

Train, Validate and Test the Model

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

In [9]:
```
# Hyperparameters

EPOCHS = 120
BATCH_SIZE = 128
rate = 0.001
```

In [10]:
```
# Training pipeline and evaluation

import tensorflow as tf

x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 43)
logits = LeNet(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)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y,
 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float
32))
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[offs
et: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

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)
    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch
_y})
        validation_accuracy = evaluate(X_valid, y_valid)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy =
{:.4f}".format(validation_accuracy))
        print()

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

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.7238

EPOCH 2 ...
Validation Accuracy = 0.7794

EPOCH 3 ...
Validation Accuracy = 0.8485

EPOCH 4 ...
Validation Accuracy = 0.8456

EPOCH 5 ...
Validation Accuracy = 0.8567

EPOCH 6 ...
Validation Accuracy = 0.8717

EPOCH 7 ...
Validation Accuracy = 0.8832

EPOCH 8 ...
Validation Accuracy = 0.8846

EPOCH 9 ...
Validation Accuracy = 0.8882

EPOCH 10 ...
Validation Accuracy = 0.8735

EPOCH 11 ...
Validation Accuracy = 0.9007

EPOCH 12 ...
Validation Accuracy = 0.8848

EPOCH 13 ...
Validation Accuracy = 0.9036

EPOCH 14 ...
Validation Accuracy = 0.8998

EPOCH 15 ...
Validation Accuracy = 0.8850

EPOCH 16 ...
Validation Accuracy = 0.8862

EPOCH 17 ...
Validation Accuracy = 0.8968

EPOCH 18 ...
Validation Accuracy = 0.8984

EPOCH 19 ...
```

```
Validation Accuracy = 0.8977

EPOCH 20 ...
Validation Accuracy = 0.9084

EPOCH 21 ...
Validation Accuracy = 0.9059

EPOCH 22 ...
Validation Accuracy = 0.9084

EPOCH 23 ...
Validation Accuracy = 0.8982

EPOCH 24 ...
Validation Accuracy = 0.9020

EPOCH 25 ...
Validation Accuracy = 0.8934

EPOCH 26 ...
Validation Accuracy = 0.9184

EPOCH 27 ...
Validation Accuracy = 0.9218

EPOCH 28 ...
Validation Accuracy = 0.9156

EPOCH 29 ...
Validation Accuracy = 0.9073

EPOCH 30 ...
Validation Accuracy = 0.9234

EPOCH 31 ...
Validation Accuracy = 0.9202

EPOCH 32 ...
Validation Accuracy = 0.9120

EPOCH 33 ...
Validation Accuracy = 0.9063

EPOCH 34 ...
Validation Accuracy = 0.9138

EPOCH 35 ...
Validation Accuracy = 0.9005

EPOCH 36 ...
Validation Accuracy = 0.9132

EPOCH 37 ...
Validation Accuracy = 0.9145

EPOCH 38 ...
```

```
Validation Accuracy = 0.9041

EPOCH 39 ...
Validation Accuracy = 0.9113

EPOCH 40 ...
Validation Accuracy = 0.9195

EPOCH 41 ...
Validation Accuracy = 0.9143

EPOCH 42 ...
Validation Accuracy = 0.9247

EPOCH 43 ...
Validation Accuracy = 0.9166

EPOCH 44 ...
Validation Accuracy = 0.9150

EPOCH 45 ...
Validation Accuracy = 0.9161

EPOCH 46 ...
Validation Accuracy = 0.9215

EPOCH 47 ...
Validation Accuracy = 0.9091

EPOCH 48 ...
Validation Accuracy = 0.9249

EPOCH 49 ...
Validation Accuracy = 0.9279

EPOCH 50 ...
Validation Accuracy = 0.9197

EPOCH 51 ...
Validation Accuracy = 0.9283

EPOCH 52 ...
Validation Accuracy = 0.9234

EPOCH 53 ...
Validation Accuracy = 0.9315

EPOCH 54 ...
Validation Accuracy = 0.9032

EPOCH 55 ...
Validation Accuracy = 0.9209

EPOCH 56 ...
Validation Accuracy = 0.9086

EPOCH 57 ...
```

```
Validation Accuracy = 0.9220

EPOCH 58 ...
Validation Accuracy = 0.9238

EPOCH 59 ...
Validation Accuracy = 0.9204

EPOCH 60 ...
Validation Accuracy = 0.9086

EPOCH 61 ...
Validation Accuracy = 0.9270

EPOCH 62 ...
Validation Accuracy = 0.9002

EPOCH 63 ...
Validation Accuracy = 0.9005

EPOCH 64 ...
Validation Accuracy = 0.9268

EPOCH 65 ...
Validation Accuracy = 0.9222

EPOCH 66 ...
Validation Accuracy = 0.9204

EPOCH 67 ...
Validation Accuracy = 0.8873

EPOCH 68 ...
Validation Accuracy = 0.9193

EPOCH 69 ...
Validation Accuracy = 0.9247

EPOCH 70 ...
Validation Accuracy = 0.9308

EPOCH 71 ...
Validation Accuracy = 0.9177

EPOCH 72 ...
Validation Accuracy = 0.9249

EPOCH 73 ...
Validation Accuracy = 0.9070

EPOCH 74 ...
Validation Accuracy = 0.9231

EPOCH 75 ...
Validation Accuracy = 0.9327

EPOCH 76 ...
```

```
Validation Accuracy = 0.9311

EPOCH 77 ...
Validation Accuracy = 0.9170

EPOCH 78 ...
Validation Accuracy = 0.9249

EPOCH 79 ...
Validation Accuracy = 0.9211

EPOCH 80 ...
Validation Accuracy = 0.9206

EPOCH 81 ...
Validation Accuracy = 0.9184

EPOCH 82 ...
Validation Accuracy = 0.9295

EPOCH 83 ...
Validation Accuracy = 0.9220

EPOCH 84 ...
Validation Accuracy = 0.9261

EPOCH 85 ...
Validation Accuracy = 0.9224

EPOCH 86 ...
Validation Accuracy = 0.9333

EPOCH 87 ...
Validation Accuracy = 0.9254

EPOCH 88 ...
Validation Accuracy = 0.9084

EPOCH 89 ...
Validation Accuracy = 0.9322

EPOCH 90 ...
Validation Accuracy = 0.9290

EPOCH 91 ...
Validation Accuracy = 0.9288

EPOCH 92 ...
Validation Accuracy = 0.9261

EPOCH 93 ...
Validation Accuracy = 0.9188

EPOCH 94 ...
Validation Accuracy = 0.9249

EPOCH 95 ...
```

```
Validation Accuracy = 0.9190

EPOCH 96 ...
Validation Accuracy = 0.9118

EPOCH 97 ...
Validation Accuracy = 0.9254

EPOCH 98 ...
Validation Accuracy = 0.9170

EPOCH 99 ...
Validation Accuracy = 0.9288

EPOCH 100 ...
Validation Accuracy = 0.9313

EPOCH 101 ...
Validation Accuracy = 0.9175

EPOCH 102 ...
Validation Accuracy = 0.9295

EPOCH 103 ...
Validation Accuracy = 0.9283

EPOCH 104 ...
Validation Accuracy = 0.9317

EPOCH 105 ...
Validation Accuracy = 0.9290

EPOCH 106 ...
Validation Accuracy = 0.9177

EPOCH 107 ...
Validation Accuracy = 0.9290

EPOCH 108 ...
Validation Accuracy = 0.9195

EPOCH 109 ...
Validation Accuracy = 0.9349

EPOCH 110 ...
Validation Accuracy = 0.9338

EPOCH 111 ...
Validation Accuracy = 0.9315

EPOCH 112 ...
Validation Accuracy = 0.9293

EPOCH 113 ...
Validation Accuracy = 0.9351

EPOCH 114 ...
```

```
Validation Accuracy = 0.9306

EPOCH 115 ...
Validation Accuracy = 0.9252

EPOCH 116 ...
Validation Accuracy = 0.9327

EPOCH 117 ...
Validation Accuracy = 0.9220

EPOCH 118 ...
Validation Accuracy = 0.9329

EPOCH 119 ...
Validation Accuracy = 0.9259

EPOCH 120 ...
Validation Accuracy = 0.9308

Model saved
```

In [11]:

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

    test_accuracy = evaluate(X_test, y_test)
    print("Test Accuracy = {:.4f}".format(test_accuracy))
```

```
Test Accuracy = 0.8964
```

Step 3: Test a Model on New Images

To give yourself more insight into how your model is working, download at least five pictures of German traffic signs from the web and use your model to predict the traffic sign type.

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

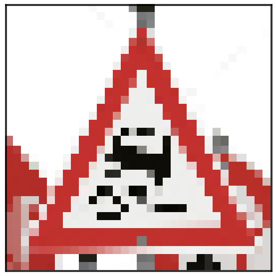
In [12]:
```
import pandas as pd
import urllib.request

def download_decode_crop_resize(url, x0=None, y0=None, x1=None, y1=None,
 w=None, h=None):
    image =
cv2.imdecode(np.frombuffer(urllib.request.urlopen(url).read(),
np.uint8), cv2.IMREAD_REDUCED_COLOR_8)

    if x0 is None:
        x0 = 0

    if y0 is None:
        y0 = 0

    if x1 is None:
        x1 = x0 + image.shape[1]

    if y1 is None:
        y1 = y0 + image.shape[0]

    if w is None:
        w = x1 - x0

    if h is None:
        h = y1 - y0

    return cv2.cvtColor(cv2.resize(image[y0:y1, x0:x1], (w, h)), cv2.COL
OR_RGB2BGR)

signnames = pd.read_csv('signnames.csv').SignName.tolist()

sources = (
    (23, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 394, 130, 646, 364),
    (42, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 176, 482, 394, 716),
    (17, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 766, 424, 970, 638),
    (36, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 874, 264, 1032, 446),
    (25, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 708, 214, 856, 378),
    (26, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 1096, 274, 1270, 454),
    (28, 'https://image.shutterstock.com/z/stock-photo-too-many-traffic-
signs-on-white-background-71581573.jpg', 1259, 274, 1396, 399)
)
```

Load and Output the Images

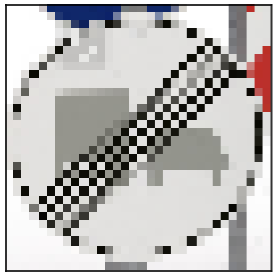
In [13]:

```
X_web = []
y_web = []

fig, axes = plt.subplots(len(sources), 1, figsize=(16, 16))

for ax, source in zip(axes.flat, sources):
    id, url, x0, y0, x1, y1 = source
    image = download_decode_crop_resize(url, x0, y0, x1, y1, 32, 32)
    gray = gray_normalized(image)
    X_web.append(gray_normalized(image))
    y_web.append(id)
    ax.set(xticks=[], yticks=[])
    ax.set_title("Sign: {}".format(signnames[id]))
    ax.imshow(image)

fig.tight_layout()
plt.show()

X_web = np.reshape(X_web, (len(sources), 32, 32, 1))
y_web = np.asarray(y_web)
```

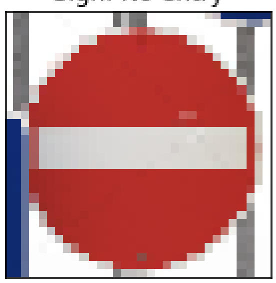
Sign: Slippery road



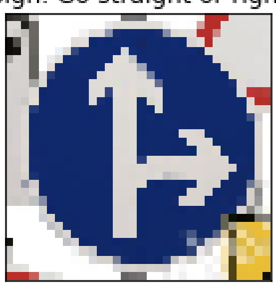
Sign: End of no passing by vehicles over 3.5 metric tons



Sign: No entry



Sign: Go straight or right



Sign: Road work

Sign: Traffic signals

Sign: Children crossing

Predict the Sign Type for Each Image

In [14]:

```
softmax = tf.nn.softmax(logits)

hits = 0

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predictions = sess.run(softmax, feed_dict={x: X_web})
    indexes = np.argmax(predictions, axis=1)

for i in range(len(y_web)):
    if y_web[i] == indexes[i]:
        hits += 1
    print("Sign = {} / Prediction = {}".format(signnames[y_web[i]], sign
names[indexes[i]]))
```

```
Sign = Slippery road / Prediction = Slippery road
Sign = End of no passing by vehicles over 3.5 metric tons / Prediction
 = Ahead only
Sign = No entry / Prediction = No entry
Sign = Go straight or right / Prediction = Go straight or right
Sign = Road work / Prediction = Road work
Sign = Traffic signals / Prediction = Traffic signals
Sign = Children crossing / Prediction = Children crossing
```

Analyze Performance

In [15]:

```
print("Accuracy = {:.0%}".format(hits/len(y_web)))
```

```
Accuracy = 86%
```

Output Top 5 Softmax Probabilities For Each Image Found on the Web

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). `tf.nn.top_k` (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k) could prove helpful here.

The example below demonstrates how tf.nn.top_k can be used to find the top k predictions for each image.

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the correspoding class ids.

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tk.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.078934
97,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

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

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

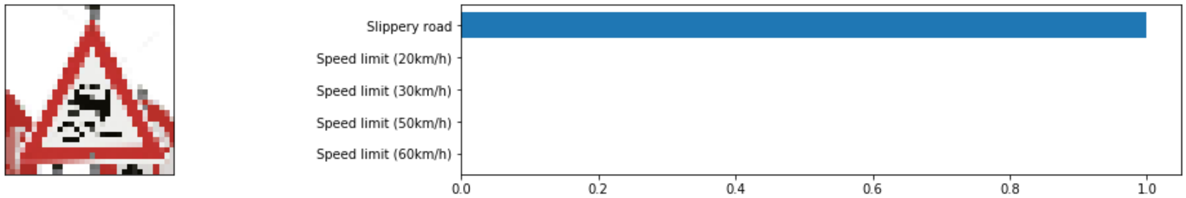
Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in `a`. You'll also notice `[3, 0, 5]` are the corresponding indices.

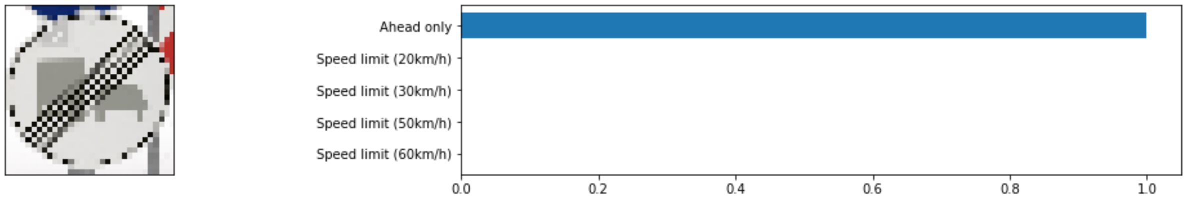
In [16]:

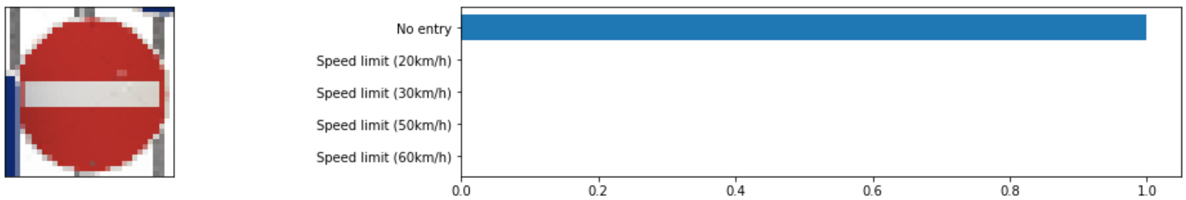
```
top_five = tf.nn.top_k(softmax, k=5)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    probs, indexes = sess.run(top_five, feed_dict={x: X_web})
    print(list(zip(indexes, probs)))
```

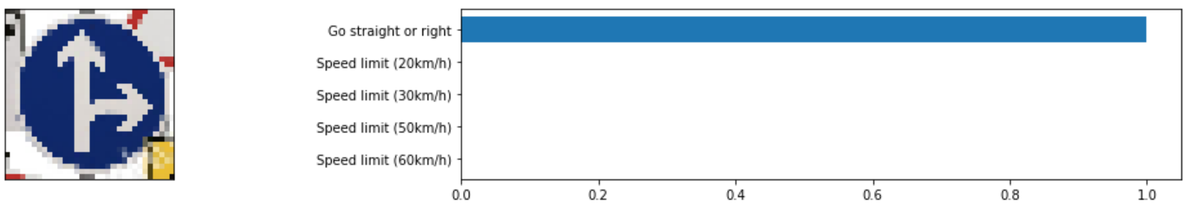
```
[(array([23,  0,  1,  2,  3], dtype=int32), array([ 1.,  0.,  0.,  0.,
0.], dtype=float32)), (array([35,  0,  1,  2,  3], dtype=int32), array
([ 1.,  0.,  0.,  0.,  0.], dtype=float32)), (array([17,  0,  1,  2,
 3], dtype=int32), array([ 1.,  0.,  0.,  0.,  0.], dtype=float32)), (a
rray([36,  0,  1,  2,  3], dtype=int32), array([ 1.,  0.,  0.,  0.,
 0.], dtype=float32)), (array([25,  0,  1,  2,  3], dtype=int32), array
([ 1.,  0.,  0.,  0.,  0.], dtype=float32)), (array([26, 24, 27, 19,
 0], dtype=int32), array([  1.00000000e+00,   4.02206064e-26,   5.46678
104e-27,
         4.45913542e-34,   0.00000000e+00], dtype=float32)), (array([2
8, 29, 30, 22, 19], dtype=int32), array([  1.00000000e+00,   4.23367764
e-12,   2.66421450e-20,
         2.09635989e-25,   1.69430730e-32], dtype=float32))]
```

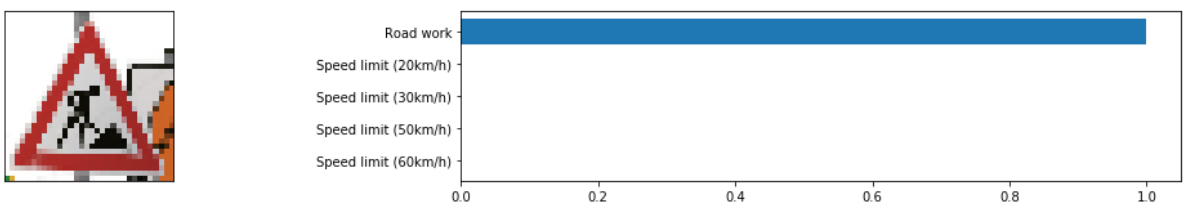
In [17]:
```
fig, axes = plt.subplots(len(sources), 2, figsize=(16, 16))

for i in range(len(sources)):
    id, url, x0, y0, x1, y1 = sources[i]
    image = download_decode_crop_resize(url, x0, y0, x1, y1, 32, 32)
    axes[i, 0].set(xticks=[], yticks=[])
    axes[i, 0].imshow(image)
    y_label = list(map(lambda x: signnames[x], indexes[i]))
    y_pos = range(len(y_label));
    axes[i, 1].barh(y_pos, probs[i])
    axes[i, 1].set_yticks(y_pos)
    axes[i, 1].set_yticklabels(y_label)
    axes[i, 1].invert_yaxis()

fig.tight_layout()
plt.show()
```

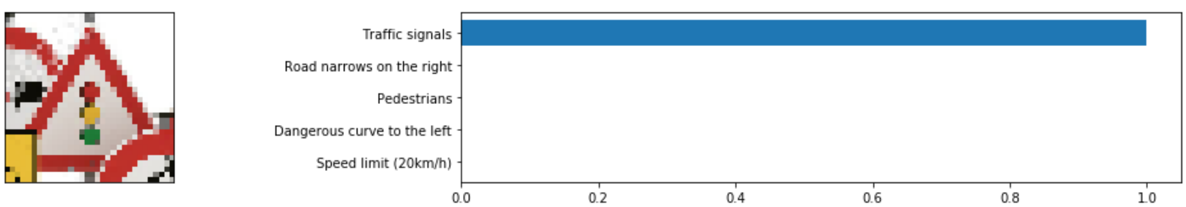


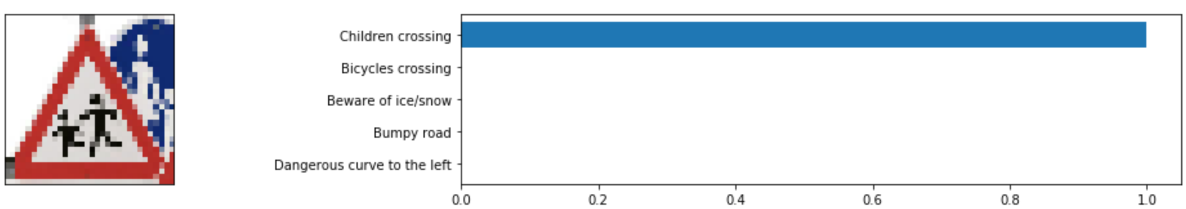












Note: Once you have completed all of the code implementations, you need to finalize your work by exporting the IPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run. You can then export the notebook by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.