Traffic_Sign_Classifier

January 21, 2017

1 Self-Driving Car Engineer Nanodegree

1.1 Deep Learning

1.2 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.

1.3 Step 0: Load The Data

In [62]:
```
# Load pickled data
import pickle

training_file = "train.p"
testing_file = "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']
```

1.4 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 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 [63]:
```
### Replace each question mark with the appropriate value.

n_train     = len(X_train)
n_test      = len(X_test)
image_shape = X_train[0].shape
n_classes   = len(set(y_train))

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

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 examples and gallery pages are a great resource for doing visualizations in Python.

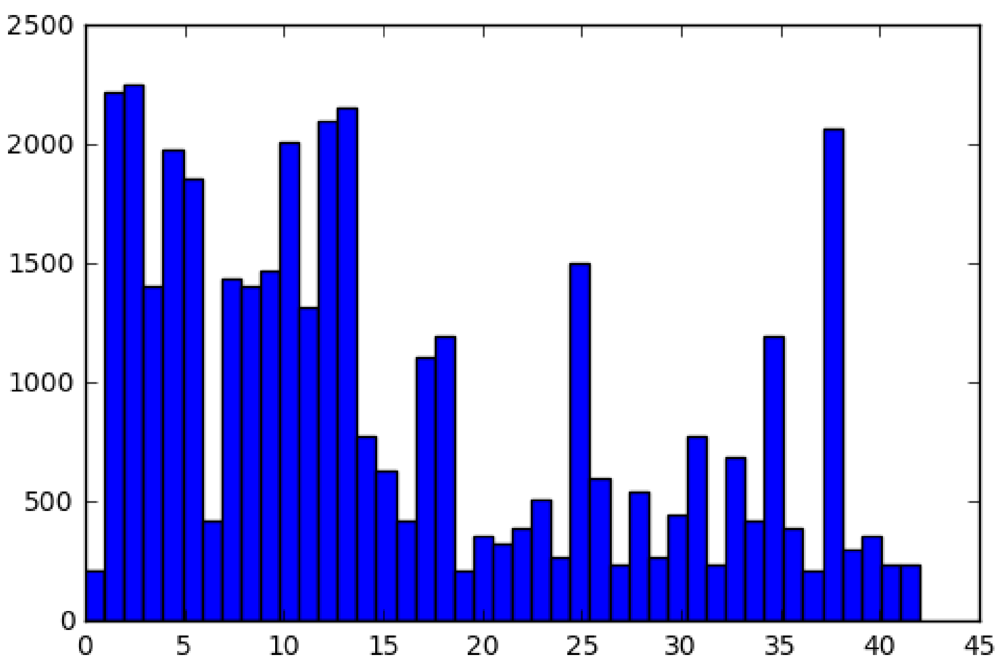
NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections.

In [64]:
```
### Data exploration visualization goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random as rand
import cv2
```

```
# Visualizations will be shown in the notebook.
%matplotlib inline

def get_labelsizes_and_start_idx(label_list):
    ref_label=label_list[0]
    first_count = len([ class_id for class_id in label_list if class_id ==
    labels=[]
    labels.append(first_count)
    sign_start_idx=[0]
    cnt=0
    for c_id in label_list:
        if ref_label != c_id:
            ref_label=c_id
            count = len([ class_id for class_id in label_list if class_id
            labels.append(count)
            sign_start_idx.append(cnt)
        cnt+=1
    return labels, sign_start_idx

labels, sign_start_idx = get_labelsizes_and_start_idx(train['labels'])

#print("Labels per each sign:", labels)
big_dataset=max(labels)
small_dataset=min(labels)
print("Biggest data set on sign:", big_dataset)
print("Smallest data set on sign:", small_dataset)

for k, sign_idx in enumerate(sign_start_idx):
    image = train['features'][sign_idx+rand.randint(0,labels[k]-1)]
    plt.subplot(6,8,k+1), plt.imshow(image)
    plt.axis('off')
```

```
Biggest data set on sign: 2250
Smallest data set on sign: 210
```


```
In [65]: plt.hist(train['labels'], bins=n_classes)
         plt.show()
```



1.5 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.

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. 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 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!

1.5.1 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 [66]: ### Preprocess the data here.
         ### Feel free to use as many code cells as needed.
         import numpy as np
         import matplotlib.pyplot as plt

         %matplotlib inline
         # some advanced preprocessing
         # https://blog.keras.io/building-powerful-image-classification-models-usi

         def greyscale(list_imgs):
             result_imgs=np.copy(list_imgs)
             #print(list_imgs[0].shape)
             result_imgs=[ cv2.cvtColor(entry, cv2.COLOR_RGB2GRAY) for entry in lis
             plt.axis('off')
             plt.imshow(result_imgs[0], cmap="gray")
             for entry in result_imgs:
                 entry.shape=(32,32,1)
             #print(result_imgs[0].shape)
             return result_imgs

         X_train_grey = greyscale(X_train)

         X_train_grey = np.asarray(X_train_grey, dtype=np.float32)
         # normalizing colors to to reduce zero mean
         X_train_grey = (X_train_grey - 128) / 128
```

1.5.2 Question 1

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

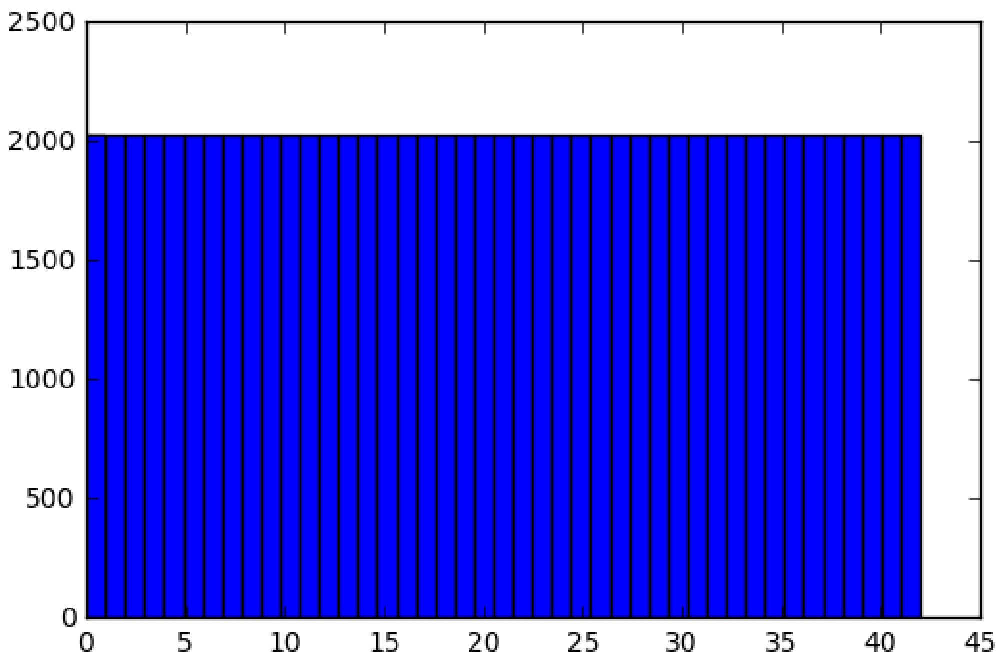
Answer: Applying greyscale conversion to all images as pointed out in paper gives better accuracy. Whereby actually it is contradicting to some other results mentioned in paper. Additonally images are normalized to get roughly zero mean and equal variance to provide the optimizer well conditioned problem space as mentioned in lectures.

```
In [67]: ### 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.

         # Would you pick remove validation from training set?
         import random

         def fill_up(features, labels, labels_per_each_sign, biggest_data, sign_sta
             x_data = []
             y_data = []
             # http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/
             rows,cols = features[0].shape[:2]
             M_t = np.float32([[1,0,random.randint(-2,2)],[0,1,random.randint(-2,2)
             M_r = cv2.getRotationMatrix2D((cols/2,rows/2),random.choice([-10,10]),
             for k, idx in enumerate(sign_start):
                 size_label = labels_per_each_sign[k]
                 diff       = biggest_data - size_label
                 if diff > 0:
```

```python
                for i in range(0, diff):
                    img = features[idx+random.randint(0,size_label-1)]
                    dst = cv2.warpAffine(img,random.choice([M_t, M_r]),(cols,r
                    dst.shape = (32,32,1)
                    x_data.append(dst)
                    y_data.append(labels[idx])
    return x_data, y_data

def split_training_set(X_train, y_train, sign_start_list, label_list, perc
    X_validation=[]
    y_validation=[]
    # taking 10% of each dataset
    to_delete=[]
    for k, sign_idx in enumerate(sign_start_list):
        # I have the feeling it is possible to do it just simple
        samples = int(label_list[k]*percent)
        indices = random.sample(range(0,samples),samples)
        for idx in indices:
            index=sign_idx + idx
            X_validation.append(X_train[index])
            y_validation.append(y_train[index])
            to_delete.append(index)
    X_train_pre=np.delete(X_train,to_delete, axis=0)
    y_train_pre=np.delete(y_train,to_delete, axis=0)
    return X_validation, y_validation, X_train_pre, y_train_pre

def generate_training_set(X_data, y_data, sign_idx, labelsizes):
    print("Data size: ",len(X_data))
    #X_train_extra, y_train_extra = fill_up(X_train_grey, y_train, labels,
    X_validation, y_validation, X_train_new, y_train_new = split_training_
    print("Validation size: ",len(X_validation))
    print("Labels size: ", len(y_validation))
    print("Training set size after extracting validation set: ", len(X_tra
    print("Labels set size after extracting validation set: ", len(y_train

    labels_new, sign_start_idx_new = get_labelsizes_and_start_idx(y_train_
    print("Labels per each sign:", labels_new)
    X_train_extra, y_train_extra = fill_up(X_train_new, y_train_new, label
    print("Training set extra: ", len(X_train_extra))
    print("Labels set extra: ", len(y_train_extra))
    X_train_pre = np.concatenate((X_train_new, X_train_extra), axis=0)
    y_train_pre = np.concatenate((y_train_new, y_train_extra), axis=0)
    return X_validation, y_validation, X_train_pre, y_train_pre

X_validation, y_validation, X_train_pre, y_train_pre = generate_training_s
print("Training set size after fill up set: ", len(X_train_pre))
print("Labels set size after fill up set: ", len(y_train_pre))
```

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

```
Data size:  39209
Validation size:  3920
Labels size:  3920
Training set size after extracting validation set:  35289
Labels set size after extracting validation set:  35289
Labels per each sign: [189, 1998, 2025, 1269, 1782, 1674, 378, 1296, 1269, 1323, 18
Training set extra:  51786
Labels set extra:  51786
Training set size after fill up set:  87075
Labels set size after fill up set:  87075
```



1.5.3 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: The training data is splitted to training set and validation set. From each sign 10% are extracted randomly for validation set. Additional data was generated by applying random translation by -2 to 2 pixels and rotation by -10 to 10 degree. Therefore we might gain some more possible real world scenarios covered. Nevertheless the training set was randomly filled to equal size of biggest data set.

In [68]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.
from tensorflow.contrib.layers import flatten

def LeNet(x, mu, sigma, classes):

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

    # 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]

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

    # 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]

    # 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, s
    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, s
    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 = num classes
```

```
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(84, classes), mean = m
    fc3_b  = tf.Variable(tf.zeros(classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits

def network_tryout(x, mu, sigma, classes):

    # conv I:32x32 O: 14x14x10
    conv1_W = tf.Variable(tf.truncated_normal(shape=(3, 3, 1, 10), mean =
    conv1_b = tf.Variable(tf.zeros(10))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALI

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

    # pooling I: 14x14x10 O: 7x7x10
    conv1 = tf.nn.avg_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1]

    # flatten I: 7x7x10 O: 490
    fc0   = flatten(conv1)

    # fullycon I: 490 O: 120
    fc1_W = tf.Variable(tf.truncated_normal(shape=(490, 120), mean = mu, s
    fc1_b = tf.Variable(tf.zeros(120))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

    # activation
    fc2    = tf.nn.relu(fc1)

    # fullycon I: 120 O: classes
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(120, classes), mean =
    fc3_b  = tf.Variable(tf.zeros(classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits

def evaluate(X_data, y_data, batch_size):
    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
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y:
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

1.5.4 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 from the classroom.

Answer: The architecture is basically the solution provided in LeNet lab described in code cell above. Only last fully connected layer output was adjusted to the number of classes of signs. Therefore it consists of two convolution layer each followed by one activation layer relu and pooling layer with max pool. The output is then flattened to 400 connections which are fully connected over three fully connected layers to output of 43 classes. The convolution layer applies a 4D to 2D transformation with stride sliding windows of [1,1,1,1] and padding algorithm VALID which is also used in all other layes. Then output is 6 feature maps of 28x28 size from an 32x32x1 input. After this applying max pool with stride 1,2,2,1 and reducing the feature maps to a size of 14x14. With additional convolution layer the 16 feature maps with size 10x10 are generated and reduced to 5x5 with max pool. Which is flattened to 400 and fully connected to 120 nodes and then to 83 and finally to 43 classes.

In [70]:
```
### Train your model here.
### Feel free to use as many code cells as needed.
from sklearn.utils import shuffle
import tensorflow as tf

# more infos https://www.tensorflow.org/api_docs/python/nn/
# https://www.tensorflow.org/api_docs/python/train/
# basic strucutre is taken from LeNet lab

EPOCHS     = 15
# based on dataset size 87075
BATCH_SIZE = 140
RATE       = 0.001
# Hyperparameters
MU         = 0
SIGMA      = 0.1

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

#logits = network_tryout(x, MU, SIGMA, n_classes)
logits = LeNet(x, MU, SIGMA, n_classes)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
# try out - several optimizers to choose
# https://www.tensorflow.org/api_docs/python/train/optimizers#Optimizer
# https://www.quora.com/Which-optimizer-in-TensorFlow-is-best-suited-for-

#global_step = tf.Variable(0, trainable=False)
```

```
    #starter_learning_rate = 0.001
    #learning_rate = tf.train.exponential_decay(starter_learning_rate, global_
    #                                            EPOCHS, 0.96, staircase=True)

    #optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    optimizer = tf.train.AdamOptimizer(learning_rate = RATE)

    #training_operation = optimizer.minimize(loss_operation, global_step=globa
    training_operation = optimizer.minimize(loss_operation)

    correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1
    accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32
    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        num_examples = len(X_train_pre)

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

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

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

```
Training...


EPOCH 1 ...
Validation Accuracy = 0.761


EPOCH 2 ...
Validation Accuracy = 0.825


EPOCH 3 ...
```

```
Validation Accuracy = 0.851


EPOCH 4 ...
Validation Accuracy = 0.843


EPOCH 5 ...
Validation Accuracy = 0.870


EPOCH 6 ...
Validation Accuracy = 0.865


EPOCH 7 ...
Validation Accuracy = 0.880


EPOCH 8 ...
Validation Accuracy = 0.875


EPOCH 9 ...
Validation Accuracy = 0.895


EPOCH 10 ...
Validation Accuracy = 0.885


EPOCH 11 ...
Validation Accuracy = 0.890


EPOCH 12 ...
Validation Accuracy = 0.891


EPOCH 13 ...
Validation Accuracy = 0.886


EPOCH 14 ...
Validation Accuracy = 0.901


EPOCH 15 ...
```

```
Validation Accuracy = 0.885

Model saved
```

1.5.5 Question 4

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

Answer: Training was done with 15 epochs and learning rate of 0.001 where higher epochs did not show segnificant improve in connection to lower learning rate. The batch size is choosen based on data size so that last batch holds enough data. Furthermore data was shuffled during training to gain more generalized training.

1.5.6 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: The choosen architecture is LeNet. The network breaks down the learning of features and samples from layer to layer more fine granular features of signs. Several other networks would be sufficient as well like GoogleNet or AlexNet. Besides this the LeNet paper results are showing accurracy of 99% which is good for this problem. This model does not achieve this accuracy due to lack of more scenarios in additional data generation.

1.5.7 Model test with existing test set

```
In [75]: # Run test
         import tensorflow as tf

         X_test_grey=np.copy(X_test)
         X_test_grey=[ cv2.cvtColor(entry, cv2.COLOR_RGB2GRAY) for entry in X_test]
         for entry in X_test_grey:
             entry.shape=(32,32,1)

         X_test_grey = np.asarray(X_test_grey, dtype=np.float32)
         X_test_grey = ( X_test_grey - 128 ) / 128

         saver = tf.train.Saver()

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

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

Test Accuracy = 0.913
```

1.6 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.

1.6.1 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 [78]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import pickle

# preconfigured images taken with own smartphone and transformed with tra
testing_file = "my_test.p"

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

#print("Original image")
#for k, image in enumerate(test['orig']):
#    plt.subplot(6,8,k+1), plt.imshow(image)

resized=test['resized'];

#X_test=[resized[10]]
X_test=[resized[0],resized[5],resized[7],resized[8],resized[10]]
#y_test=[18]
y_test=[25,17,38,12,18]

print("Choosen images")
for k, image in enumerate(X_test):
    plt.subplot(6,8,k+1), plt.imshow(image)
    plt.axis('off')

resized=[ cv2.cvtColor(entry, cv2.COLOR_RGB2GRAY) for entry in resized]
for entry in resized:
    entry.shape=(32,32,1)

X_test=[resized[0],resized[5],resized[7],resized[8],resized[10]]
```

```
X_test = np.asarray(X_test, dtype=np.float32)
X_test = ( X_test - 128 ) / 128
```

```
Choosen images
```

1.6.2 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: Some difficulties are day/night conditions, light reflection, perspective, too blurry, bad resolution, detecting signs in big distance, stickers on sign like the keep right sign.

```
In [79]: ### Run the predictions here.
         ### Feel free to use as many code cells as needed.
         import tensorflow as tf

         saver = tf.train.Saver()

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

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

```
Test Accuracy = 0.800
```

1.6.3 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.

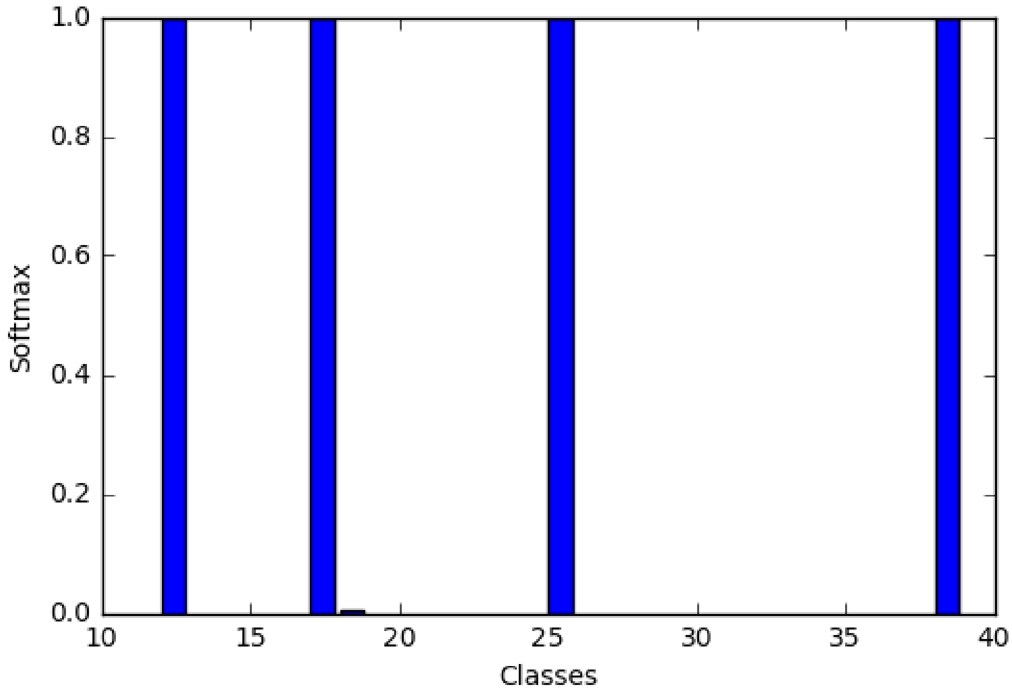
NOTE: *You could check the accuracy manually by using `signnames.csv` (same directory). This file has a mapping from the class id (0-42) to the corresponding sign name. So, you could take the class id the model outputs, lookup the name in `signnames.csv` and see if it matches the sign from the image.*

Answer:

The accuracy ist 80%. Only the "general caution" sign was not recognized. The result is below the accuracy of test 91% which is hard to compare with data set of 5 signs. Considering the last sign without text would end up in 100% but would not reflect the accuracy of the model.

In [92]:
```
### Visualize the softmax probabilities here.
### Feel free to use as many code cells as needed.
#print(correct_prediction)
import tensorflow as tf

saver = tf.train.Saver()

softmax=tf.nn.softmax(logits)
pred=tf.nn.top_k(softmax, 5)
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    out = sess.run(pred, feed_dict= {x: X_test})
    print(out)
    plt.bar(y_test, [ out[0][0][0], out[0][1][0], out[0][2][0], out[0][3][
    plt.xlabel("Classes")
    plt.ylabel("Softmax")
```

```
TopKV2(values=array([[  9.99922276e-01,   7.77067980e-05,   3.35195227e-09,
          1.37205830e-10,   1.78495291e-11],
       [  1.00000000e+00,   1.24965854e-14,   7.44584101e-16,
          4.43229630e-16,   4.16247052e-16],
       [  1.00000000e+00,   5.67139738e-11,   4.92119874e-14,
          6.61910879e-16,   3.45071720e-16],
       [  1.00000000e+00,   4.55334748e-09,   7.18972243e-11,
          4.94964451e-11,   3.26392841e-12],
       [  9.94955003e-01,   4.94238408e-03,   4.32880661e-05,
          3.42314015e-05,   2.30762453e-05]], dtype=float32), indices=array([[25, 2
       [17, 39, 34, 33, 19],
       [38,  3, 23, 20, 34],
       [12, 27, 41, 32, 28],
       [31, 18, 11, 23, 26]], dtype=int32))
```



1.6.4 Question 8

*Use the model's softmax probabilities to visualize the **certainty** of its predictions, [`tf.nn.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: The model was certain about the "road work", "no entry", "keep right" and "priority road" sign even the "keep right" sign has some obstractions on it. Last but not least the "general caution" sing additionally with some text below was only predicted with k=2. This is probably caused by the poor quality of the image and confusion by the text label.