# 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 1: Dataset Exploration

Visualize the German Traffic Signs Dataset. This is open ended, some suggestions include: plotting traffic signs images, plotting the count of each sign, etc. Be creative!

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

- features -> the images pixel values, (width, height, channels)
- labels -> the label of the traffic sign
- sizes -> the original width and height of the image, (width, height)
- coords -> coordinates of a bounding box around the sign in the image, (x1, y1, x2, y2). Based the original image (not the resized version).

In [1]:
```
# Load pickled data
import pickle

# TODO: fill this in based on where you saved the training and testing data
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']
```

In [2]:
```
### To start off let's do a basic data summary.
import numpy as np
from PIL import Image
from IPython.display import display

# TODO: number of training examples
n_train = y_train.shape[0]

# TODO: number of testing examples
n_test = y_test.shape[0]

# TODO: what's the shape of an image?
image_shape = X_train[0].shape

# TODO: how many classes are in the dataset
n_classes = np.unique(y_train).shape[0]

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)
print("Min & Max class =", np.min(y_train), np.max(y_train))
for i in range(n_classes):
    print("Class:", i, "Train Count:",np.sum((y_train == i).astype(int)), "Test Count", np.sum((y_tes
t==i).astype(int)))
    firstim = X_test[np.where(y_test==i)[0][0]].copy()
    firstim = Image.fromarray(firstim.astype(np.uint8), 'RGB')
    display(firstim)
```

```
Number of training examples = 39209
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
Min & Max class = 0 42
Class: 0 Train Count: 210 Test Count 60
```

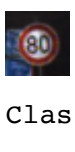
```
Class: 1 Train Count: 2220 Test Count 720
```

```
Class: 2 Train Count: 2250 Test Count 750
```

```
Class: 3 Train Count: 1410 Test Count 450
```

```
Class: 4 Train Count: 1980 Test Count 660
```

```
Class: 5 Train Count: 1860 Test Count 630
```



```
Class: 6 Train Count: 420 Test Count 150
```

```
Class: 7 Train Count: 1440 Test Count 450
```

```
Class: 8 Train Count: 1410 Test Count 450
```

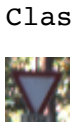
```
Class: 9 Train Count: 1470 Test Count 480
```

```
Class: 10 Train Count: 2010 Test Count 660
```

```
Class: 11 Train Count: 1320 Test Count 420
```

```
Class: 12 Train Count: 2100 Test Count 690
```

```
Class: 13 Train Count: 2160 Test Count 720
```



```
Class: 14 Train Count: 780 Test Count 270
```

```
Class: 15 Train Count: 630 Test Count 210
```

```
Class: 16 Train Count: 420 Test Count 150
```

```
Class: 17 Train Count: 1110 Test Count 360
```

```
Class: 18 Train Count: 1200 Test Count 390
```

```
Class: 19 Train Count: 210 Test Count 60
```

```
Class: 20 Train Count: 360 Test Count 90
```

Class: 21 Train Count: 330 Test Count 90

Class: 22 Train Count: 390 Test Count 120

Class: 23 Train Count: 510 Test Count 150

Class: 24 Train Count: 270 Test Count 90

Class: 25 Train Count: 1500 Test Count 480

Class: 26 Train Count: 600 Test Count 180

Class: 27 Train Count: 240 Test Count 60

Class: 28 Train Count: 540 Test Count 150

Class: 29 Train Count: 270 Test Count 90

Class: 30 Train Count: 450 Test Count 150

Class: 31 Train Count: 780 Test Count 270

Class: 32 Train Count: 240 Test Count 60

Class: 33 Train Count: 689 Test Count 210

Class: 34 Train Count: 420 Test Count 120

Class: 35 Train Count: 1200 Test Count 390

Class: 36 Train Count: 390 Test Count 120

Class: 37 Train Count: 210 Test Count 60

Class: 38 Train Count: 2070 Test Count 690

Class: 39 Train Count: 300 Test Count 90

Class: 40 Train Count: 360 Test Count 90

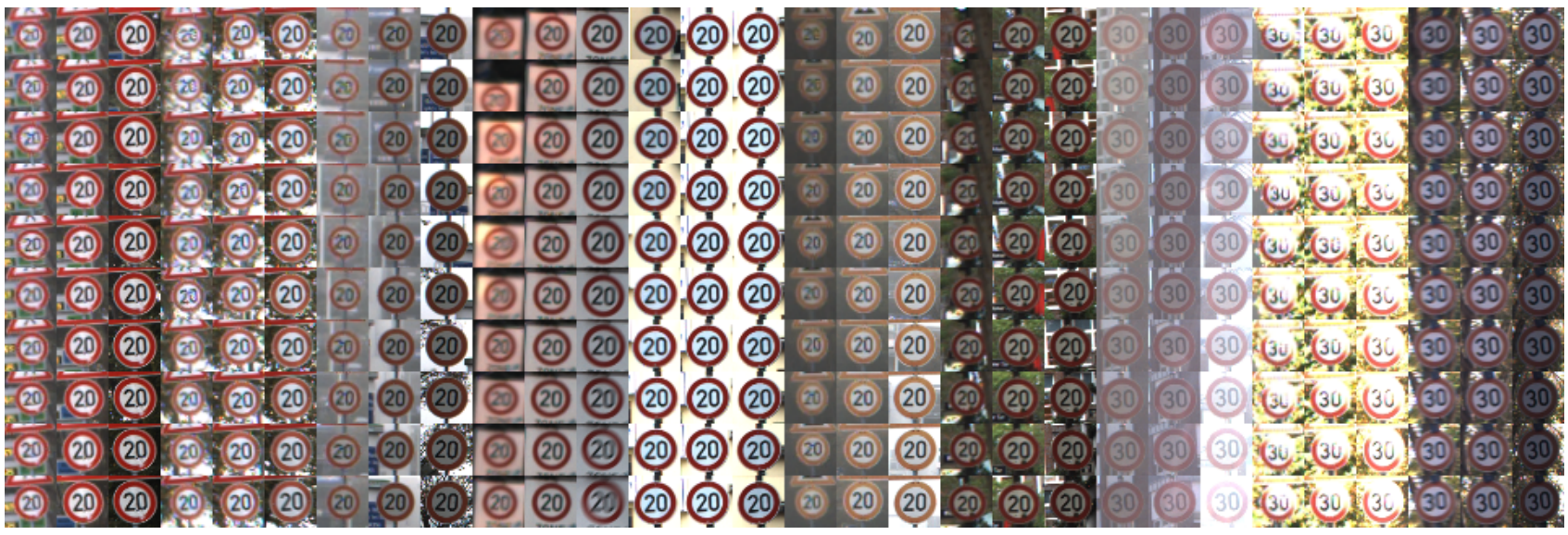
Class: 41 Train Count: 240 Test Count 60

```
Class: 42 Train Count: 240 Test Count 90
```

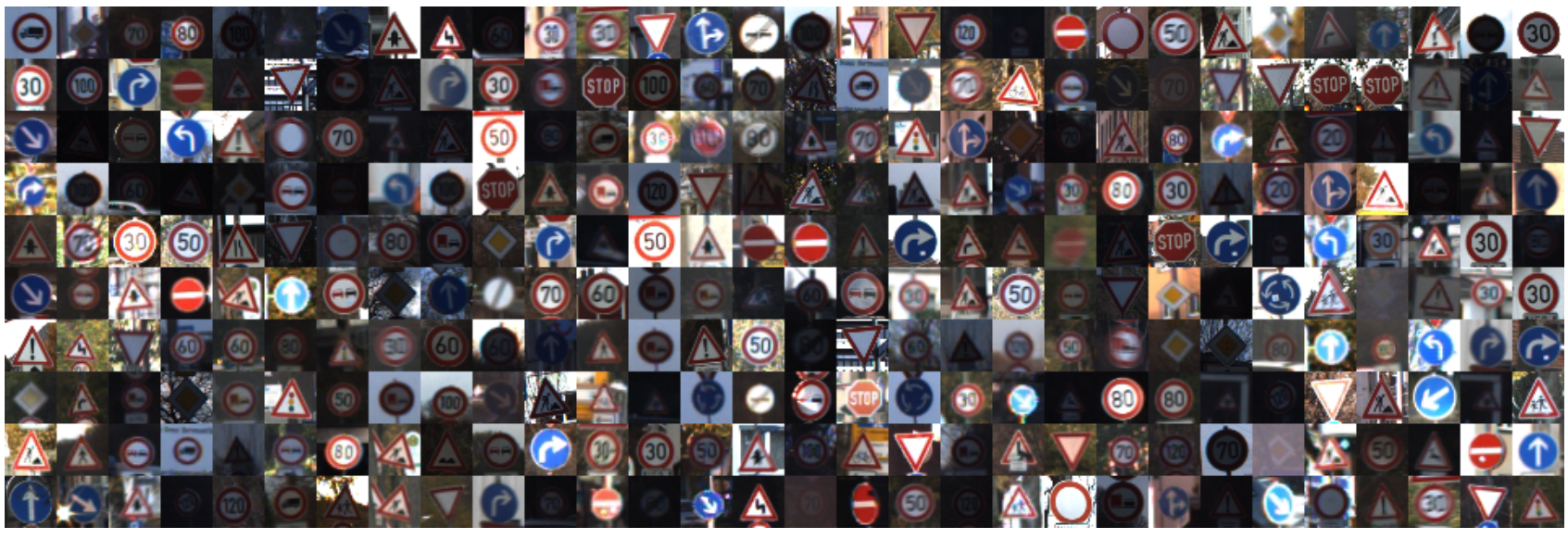
In [5]:
```
### Data exploration visualization goes here.
### Feel free to use as many code cells as needed.
import random as rand
rand.seed(113)

def showImages(xr, yr, array, indices):
    iData = np.zeros((xr*32, yr*32, 3), dtype=int)
    for (i,ix) in enumerate(indices):
        x,y = i%xr*32, int(i/xr)*32
        iData[x:x+32,y:y+32,:] = array[ix]
    display(Image.fromarray(iData.astype(np.uint8), 'RGB'))

xr,yr = 10,30
total = xr*yr
randTrainIndices = rand.sample(range(39209), total)
orderedIndices = list(range(total))

print("Unique images from training set:", np.unique(randTrainIndices).shape[0])
showImages(xr,yr,X_train,randTrainIndices)
print("Ordered images from training set:", total)
showImages(xr,yr,X_train,orderedIndices)
print("Ordered images from test set:", total)
showImages(xr,yr,X_test,orderedIndices)
```

```
Unique images from training set: 300
```

```
Ordered images from training set: 300
```



```
Ordered images from test set: 300
```



---

## 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:

- Your model can be derived from a deep feedforward net or a deep convolutional network.
- Play around preprocessing techniques (normalization, rgb to grayscale, etc)
- Number of examples per label (some have more than others).
- Generate fake data.

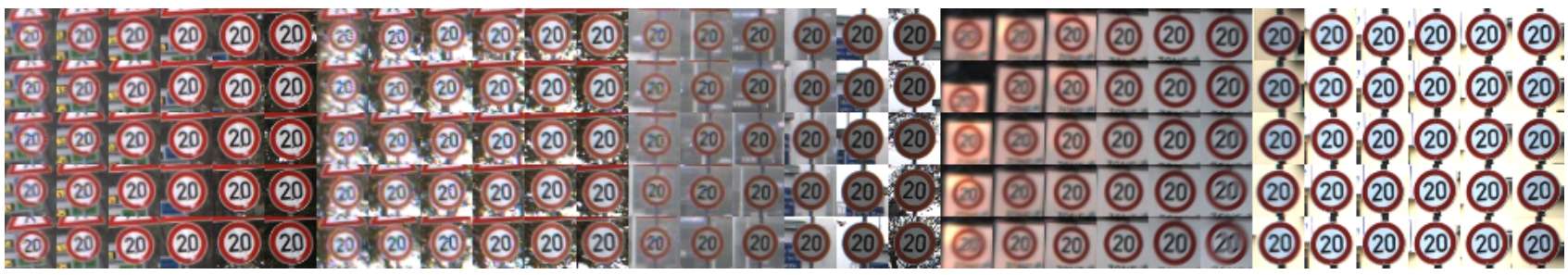
Here is an example of a published baseline model on this problem (http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

### 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 [6]:
```
### Preprocess the data here.

#Denormalize per-image, back to [0,255] integer array
def denormalize(array):
    array = array.copy()
    for (i,x) in enumerate(array.copy()):
        x = x.copy().reshape(32*32,3)
        x = x - np.min(x, axis=0)
        x = x.astype(np.float32) / (np.max(x, axis=0)  + 0.0000001)
        x = x * 255.0
        x = np.clip(x.astype(int), 0, 255)
        array[i] = x.reshape(32,32,3)
    return array

#Standardize per-image values, normalize with mean 0, range somewhere between [-1,1]
def preprocess(array):
    array = np.clip(array.copy(), 2, 253).astype(np.float32)

    for (i,x) in enumerate(array.copy()):
        cx = x[10:22,10:22,:].copy().reshape(12*12,3).astype(np.float32)
        stdx = 2.0 * np.std(cx, axis=0)
        meanx = np.mean(cx, axis=0)
        x = x.copy().reshape(32*32,3)
        x = np.clip(x,meanx-stdx,meanx+stdx)

        x = x - np.min(x, axis=None)
        x = x.astype(np.float32) / (np.max(x, axis=None)  + 0.0000001)
        x = x - np.mean(x, axis=None)

        array[i] = x.reshape(32,32,3)

    return array

#Process
print("normalizing test images")
NX_test = preprocess(X_test)
print("normalizing train images")
NX_train = preprocess(X_train)

#Visualize results
print("visualizing")
xr,yr = 5,30
total = xr*yr
orderedIndices = list(range(total))
VTRAIN = denormalize(NX_train[:total])
VTEST = denormalize(NX_test[:total])

print("Ordered images from training set:", total)
showImages(xr,yr,X_train,orderedIndices)
print("Processed")
showImages(xr,yr,VTRAIN,orderedIndices)
print("Ordered images from test set:", total)
showImages(xr,yr,X_test,orderedIndices)
print("Processed")
showImages(xr,yr,VTEST,orderedIndices)
```

```
normalizing test images
normalizing train images
visualizing
Ordered images from training set: 150
```



```
Processed
```

```
Ordered images from test set: 150
```

```
Processed
```

## Question 1

*Describe the techniques used to preprocess the data:*

I process per Image, not per feature, to fix images that were too dark or bright, and normalize mean of features to make training NN go more quickly.
Take the central image area (likely chunk of the sign), and get the mean and standard deviation values per color.
Then I clip the entire image's colors to be within two std's from the mean of that central area's colors.
This makes any extra-light or extra-dark areas on the outside of the image closer to the sign's brightness/colors.
We can see above that it's even easier for a human to identify.
Then I normalize the values (not per color) to be of range [0,1] by subtracting min and dividing max.
Then I subtract the mean value (not per color) to bring the mean overal brightness of the image to 0.
Bringing the data in this range will make the NN with tanh activation functions work more smoothly.

I provide a denormalization that just moves the range back to 0,255 so I can visualize the processing results.

**Answer:**

In [7]:
```
### Generate data additional (if you want to!)

### and split the data into training/validation/testing sets here.
from sklearn.model_selection import train_test_split
twentyPercent = int(y_train.shape[0] / 5)
SX_train, SX_val, SY_train, SY_val = \
    train_test_split(NX_train, y_train, test_size=0.1, random_state=11, stratify=y_train)

### Feel free to use as many code cells as needed.
print(SX_train.shape, SX_val.shape, SY_train.shape, SY_val.shape)
```

```
(35288, 32, 32, 3) (3921, 32, 32, 3) (35288,) (3921,)
```

## Question 2

*Describe how you set up the training, validation and testing data for your model. If you generated additional data, why?*

Why Not Generate More Data?: Well, Flipping LR or TB, Cropping, or Rotating would be my usual methods, But some of the signs have spatially significant non-mirrorable/rotatable features, and the signs are boxed already. E.G. 20/50 or arrows pointing different angles, so I'm going to not do this part for now, and we have lots of data...

Testing data, i kept as being only from the test dataset, so it's comparable in the end to other benchmarks. This is my don't touch till the end dataset.

Validation data, i used only about 4k values, 10% of the training dataset, as it is enough data to show some significance in the resulting accuracy.

The rest of the data I use for training, I didn't do full k-fold cross validation, for sake of time.

I utilized the random state and I also stratified to maintain the percentage distribution split of class labels between train/validation with diverse examples per class.

**Answer:**

In [14]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.
import tensorflow as tf

train_lambda = 0.0003
beta = 0.25
drop = 0.15

lowest_valid_score = 99999999.0
current_patience = 0
TRAIN_COST = []
VALID_COST = []

x = tf.placeholder(tf.float32, shape=[ None, 32, 32, 3 ])
y = tf.placeholder(tf.int32, shape=[ None ])
keep = tf.placeholder("float")
rand_seed = 1

weight_matrix = []
bias_matrix = []
strides_matrix = []
padding_matrix = []

def make_tf_variable( shape ):
        global rand_seed
        rand_seed += 2
        return tf.Variable(tf.truncated_normal(shape, mean=0.0, stddev=0.2, seed=rand_seed))

def add_layer( shape, strides=[1,2,2,1], padding='VALID'):
        global weight_matrix
        global bias_matrix
        global strides_matrix
        global padding_matrix
        weight_matrix.append( make_tf_variable( shape ) )
        bias_matrix.append(   make_tf_variable( shape[-1:] ) )
        strides_matrix.append( strides )
        padding_matrix.append( padding )

def convolve(logit, index):
        global weight_matrix
        global bias_matrix
        global strides_matrix
        global padding_matrix
        cd = tf.nn.conv2d(logit, weight_matrix[index], strides=strides_matrix[index], padding=padding
_matrix[index])
        cd = tf.add( cd, bias_matrix[index] )
        return tf.nn.tanh(cd)

add_layer([3,3,3,27]) #15x15
add_layer([3,3,27,100], strides=[1,1,1,1], padding='SAME') #15x15
add_layer([3,3,100,200]) #7x7
add_layer([3,3,200,400]) #3x3
convs = len(weight_matrix)
dimmy = 3*3*400
add_layer([dimmy, 43])
add_layer([43,43])

full_matrix = []
full_matrix.append(x)

for i in range(convs-1):
        tc = convolve(full_matrix[i], i)
        if (i > 0 and drop > 0.0001):
                tc = tf.nn.dropout(tc, keep)
        full_matrix.append(tc)

tc = convolve(full_matrix[convs-1], convs-1)
```

```
tc = tf.nn.dropout(tc, keep)
fa = tf.reshape(tc, [-1, dimmy])
full_matrix.append(fa)

for i in range(convs, len(weight_matrix) - 1):
        (weight,bias) = (weight_matrix[i], bias_matrix[i])
        a = tf.nn.tanh( tf.add( tf.matmul( full_matrix[i], weight ) , bias ) )
        full_matrix.append(a)

last_layer = len(weight_matrix) - 1
y_ = tf.add( tf.matmul( full_matrix[last_layer], weight_matrix[last_layer]) , bias_matrix[last_layer]
 )

correct_prediction = tf.equal(y, tf.cast(tf.argmax(y_,1), tf.int32))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(y_,y))

reg = tf.constant(0.0)
if (beta > 0):
    for i in range(len(weight_matrix)):
            reg = reg + (beta * tf.nn.l2_loss(tf.reduce_mean(weight_matrix[i])))
            reg = reg + (beta * tf.nn.l2_loss(tf.reduce_mean(bias_matrix[i])))

cost = loss + reg
train_step = tf.train.AdamOptimizer(train_lambda).minimize(cost)
init = tf.global_variables_initializer()
```

## 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:**

Using a deep conv-net with fc end layers and softmax with cross entropy cost, here is the shape:
INPUT: W: 32, H: 32, D: 3 ( output shape: 32x32x3 )
CONV: FILTERS: 3x3x27, STRIDES: 2x2, PADDING: NONE ( output shape: 15x15x27 ) -> TANH -> 15% DROPOUT
CONV: FILTERS: 3x3x100, STRIDES: 1x1, PADDING: 1 ( output shape: 15x15x100 ) -> TANH -> 15% DROPOUT
CONV: FILTERS: 3x3x200, STRIDES: 2x2, PADDING: NONE ( output shape: 7x7x200 ) -> TANH -> 15% DROPOUT
CONV: FILTERS: 3x3x400, STRIDES: 2x2, PADDING: NONE ( output shape: 3x3x400 ) -> TANH -> 15% DROPOUT -> flatten
FCON: NODES: 43 -> TANH
FCON: NODES: 43 -> SOFTMAX -> CROSS-ENTROPY + WEIGHT-REGULARIZTION -> AdamOptimizer

In [15]:
```
### Train your model here.
### Feel free to use as many code cells as needed.
from sklearn.utils import shuffle

sess = tf.Session()
sess.run(init)

loops = 160000
batch_size = 2000
print_wait = 20
estop_patience = 250

bi = -batch_size
tot_i = SX_train.shape[0]
def get_train_batch():
        global bi
        global tot_i
        global rand_seed
        global SX_train
        global SY_train
        bi += batch_size
        if (bi+batch_size > tot_i):
                rand_seed += 2
                bi = 0
                SX_train, SY_train = shuffle(SX_train, SY_train, random_state=rand_seed)
        bix = SX_train[bi:(bi + batch_size)]
        biy = SY_train[bi:(bi + batch_size)]
        return (bix.copy(), biy.copy())
```

```
def save_results():
    for (i,w) in enumerate(weight_matrix):
            np.save(t,sess.run(w))
    for (i,b) in enumerate(bias_matrix):
            np.save('MODEL_b'+str(i)+'.npy',sess.run(b))

highest_vac = 0.0
for i in range(loops):
        if (i%print_wait == 0):
                (tx,ty) = get_train_batch()
                (_,tloss,tacc) = sess.run([train_step, loss, accuracy], feed_dict={ x: tx, y: ty, kee
p: (1.0-drop) })
                (vloss,vacc,vreg) = sess.run([loss, accuracy, reg], feed_dict={ x: SX_val, y: SY_val,
 keep: 1.0 })
                smark = "  *" if (vacc > highest_vac) else " "
                print('Trainloss: ', tloss, '\t tacc:', tacc, '\t val_loss: ', vloss, '\t vacc: ', va
cc, '\t vreg:', vreg, smark)
                if (vacc > highest_vac):
                        save_results()
                        highest_vac = vacc
                        current_patience = 0
                else:
                        current_patience += 1
                if (current_patience > estop_patience):
                        break #EARLY STOPPING
        else:
                (tx,ty) = get_train_batch()
                sess.run(train_step, feed_dict={ x: tx, y: ty, keep: (1.0-drop) })

print('finished training')
sess.close()
```

In [38]:
```
#LETS CHECK OUT THE TEST SET PERFORMANCE NOW THAT WE'RE DONE FINETUNING THE TRAINING
import numpy as np
import tensorflow as tf

def TestArray(XX,YY):
    global weight_matrix
    global bias_matrix
    global strides_matrix
    global padding_matrix

    x = tf.placeholder(tf.float32, shape=[ None, 32, 32, 3 ])
    y = tf.placeholder(tf.int32, shape=[ None ])

    weight_matrix = []
    bias_matrix = []
    strides_matrix = []
    padding_matrix = []

    def add_layer( index, strides=[1,2,2,1], padding='VALID'):
            global weight_matrix
            global bias_matrix
            global strides_matrix
            global padding_matrix
            weight_matrix.append( tf.constant( np.load('MODEL_w'+str(index)+'.npy') ) )
            bias_matrix.append(   tf.constant( np.load('MODEL_b'+str(index)+'.npy')) )
            strides_matrix.append( strides )
            padding_matrix.append( padding )

    def convolve(logit, index):
            global weight_matrix
            global bias_matrix
            global strides_matrix
            global padding_matrix
            cd = tf.nn.conv2d(logit, weight_matrix[index], strides=strides_matrix[index], padding=pad
ding_matrix[index])
            cd = tf.add( cd, bias_matrix[index] )
            return tf.nn.tanh(cd)

    add_layer(0) #15x15
    add_layer(1, strides=[1,1,1,1], padding='SAME') #15x15
    add_layer(2) #7x7
    add_layer(3) #3x3
    convs = len(weight_matrix)
    dimmy = 3*3*400
    add_layer(4)
    add_layer(5)

    full_matrix = []
    full_matrix.append(x)

    for i in range(convs-1):
            tc = convolve(full_matrix[i], i)
            full_matrix.append(tc)

    tc = convolve(full_matrix[convs-1], convs-1)
    fa = tf.reshape(tc, [-1, dimmy])
    full_matrix.append(fa)

    for i in range(convs, len(weight_matrix) - 1):
            (weight,bias) = (weight_matrix[i], bias_matrix[i])
            a = tf.nn.tanh( tf.add( tf.matmul( full_matrix[i], weight ) , bias ) )
            full_matrix.append(a)

    last_layer = len(weight_matrix) - 1
    y_ = tf.add( tf.matmul( full_matrix[last_layer], weight_matrix[last_layer]) , bias_matrix[last_la
yer] )
    y_ = tf.nn.softmax(y_)
    classes = tf.nn.top_k(y_, k=5)

    correct_prediction = tf.equal(y, tf.cast(tf.argmax(y_,1), tf.int32))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)

    (testacc,topclasses) = sess.run([accuracy, classes], feed_dict={ x: XX, y: YY, keep: 1.0 })

    sess.close()
    return (testacc,topclasses)

print("FINAL TEST SET PERFORMANCE: ", TestArray(NX_test,y_test)[0])
```

```
FINAL TEST SET PERFORMANCE:  0.943785
```

## Question 4

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

**Answer:**

Optimizer: AdamOptimizer
Learning Rate: 0.0003
Regularization: 0.25 * SUM[over every layer]( L2Loss(Average( weights )) + L2Loss(Average( biases )) )
Dropout for Conv Layers (except first conv layer, which has no dropout): 15%
Weight and Bias Initiliazation: Truncated Normal, Mean: 0, Std: 0.2
Batch Size: 2000
Loops: Up to 160000 minibatches
Check Validation Scores Every: 20 Loops
Epochs: 35288/2000 = 17.644 loops/epoch, display every 20, so each training score print above is a bit over 1 epoch
Early Stopping if Validation Scores Didnt improve after: 250 Val Score Checks
Only Save Weights to numpy array if new lowest validation accuracy is reached
Shuffled training data after each epoch

## Question 5

*What approach did you take in coming up with a solution to this problem?*

**Answer:**

Used conv layer with FC end layers and softmax with cross-entropy as commonly seen approach to CV classification.
Convolutional layers help to identify spatial sub-features of images and are thought to work like the visual layers of the human visual processing system. They have been performing at state of the art levels on many of the current challenges with computer vision. (see image below regarding convolutional layer visualizations learning spatial features, Image source: Maurice Peemen, http://parse.ele.tue.nl/mpeemen (http://parse.ele.tue.nl/mpeemen) Further visualizations of similar techniques can be viewed at: https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/ (https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/))

Then I experimented with the layer sizes and hyperparameters to make training and validation accuracy go up.
If Training accuracy couldn't be learned, I made the model bigger, as it was likely not big enough to learn. If Validation score didn't improve with training score, made regularization heavier, as it was likely overfitting.

Future work: could try more data, grayscale, maxpooling, batch-normalization, inception nodes, res-netting, transfer-learning, etc..

In [201]:
```
#Cool Visualizing The Learned Filter Weights of LAYER 0 : Conv - [3,3,3,27]
weights0 = np.load('MODEL_w0.npy').swapaxes(0,3)
xr,yr = 3,9
iData = np.zeros((xr*3, yr*3, 3), dtype=np.float32)
for i in range(27):
    x,y = i%xr*3, int(i/xr)*3
    iData[x:x+3,y:y+3,:] = weights0[i,:,:,:]
iData = iData - np.min(iData, axis=None)
iData = iData / (np.max(iData, axis=None) + 0.0000001)
iData = np.clip(iData * 255.0, 0, 255).astype(int)
display(Image.fromarray(iData.astype(np.uint8), 'RGB').resize((9*100,3*100),Image.ANTIALIAS))
```

## 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 [39]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import os
pictos = np.zeros((100,7*100,3), dtype=np.int)
pictData = np.zeros((7, 32, 32, 3), dtype=np.int)
for (i,f) in enumerate(np.array(['MYSTOPS/'+f for f in os.listdir('MYSTOPS') if f.endswith('.jpeg')])
[:7]):
    picture = Image.open(f).convert('RGB').resize((100,100), Image.ANTIALIAS)
    shrunken = picture.copy().resize((32,32), Image.ANTIALIAS)
    pictData[i,:,:,:] = \
        np.array(list(shrunken.getdata())).reshape(32,32,3)
    pictos[:,i*100:i*100+100,:] = \
        np.array(list(picture.getdata())).reshape(100,100,3)

np.save('mypictdata.npy', pictData)
im = Image.fromarray(pictos.astype(np.uint8), 'RGB')
display(im)
```


## 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 would be helpful to plot the images in the notebook.*

**Answer:** I will use all of the above 7 non-dataset stop signs as candidate traffic sign images.
They should be hopefully be classifiable by a robust algorithm since the dataset stop signs are of similar type.
Some difficulties may be Angles, Lights, Different background, Non-Centered, and other abnormalities that deviate from the dataset biases.

In [56]:
```
### Run the predictions here.
### Feel free to use as many code cells as needed.
def finaltest(mytest, printdata=False):
    #Process
    PTEST = preprocess(mytest)

    #Visualize results
    if (printdata):
        xr,yr = 1,7
        total = xr*yr
        orderedIndices = list(range(total))
        VTEST = denormalize(PTEST[:total])
        print("Final images from real test:", total)
        showImages(xr,yr,mytest,orderedIndices)
        print("Final images from real test Processed")
        showImages(xr,yr,VTEST,orderedIndices)

    #Run on Model
    (testacc,topclasses) = TestArray(PTEST, np.array([14,14,14,14,14,14,14]))
    if (printdata):
        print ("small real test accuracy %:")
        print ("\t",testacc*100)
        print ("")
        print ("top choice confidence %:")
        print (np.round(topclasses[0][:,0] * 100).astype(int))
        print ("")
        print ("top-5 choice of classes:")
        print ( topclasses[1] )
    return (testacc,topclasses)

images = np.load('mypictdata.npy')
_ = finaltest(images, printdata=True)
```

```
Final images from real test: 7
```

```
Final images from real test Processed
```

```
small real test accuracy %:
         57.1428596973

top choice confidence %:
[ 99  59  69  85 100  94 100]

top-5 choice of classes:
[[ 1  0 14 17 13]
 [ 1 14 17  0 25]
 [14  3 10  9 17]
 [14 17  1 25 22]
 [14 22 15 25  1]
 [ 1 17  0 14  4]
 [14  1 15 17  9]]
```

## Question 7

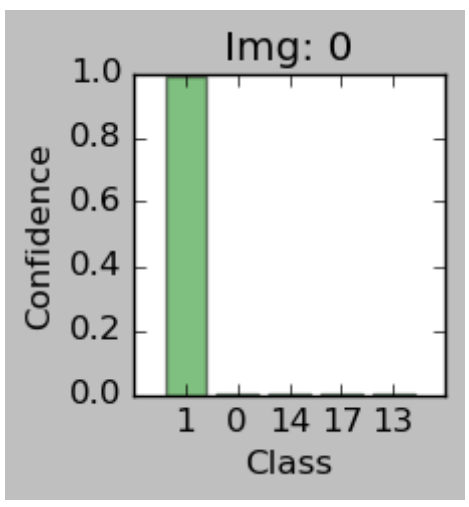
*Is your model able to perform equally well on captured pictures when compared to testing on the dataset?*

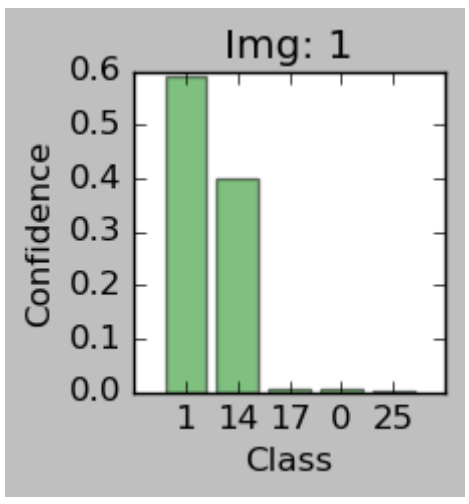
**Answer:**

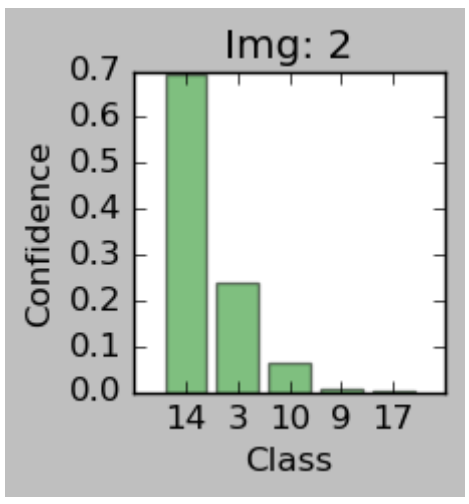
NO! 94.38% accuracy on test set, but only 57.1429 accuracy on this data that seems easy for a human to get right!

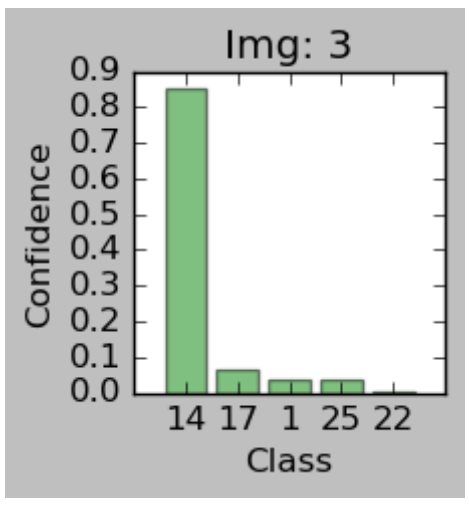
In [202]:
```
### Visualize the softmax probabilities here.
### Feel free to use as many code cells as needed.
images = np.load('mypictdata.npy')
_,topk = finaltest(images, printdata=False)

import matplotlib.pyplot as plt; plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt

showImages(1,7,images,range(7))

for i in range(7):
    plt.figure(figsize=(2,2))
    plt.bar(np.arange(len(topk[1][i])), topk[0][i], align='center', alpha=0.5, color='green')
    plt.xticks(np.arange(len(topk[1][i])), topk[1][i])
    plt.ylabel('Confidence')
    plt.xlabel('Class')
    plt.title('Img: ' +str(i))
plt.show()
```

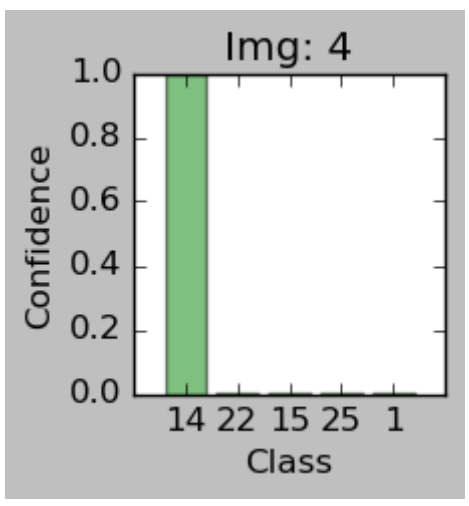

STOP
STOP
STOP
STOP
STOP
STOP
STOP

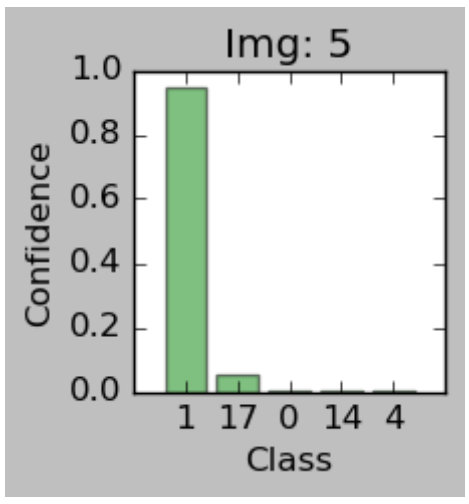

Img: 0
Confidence
1.0
0.8
0.6
0.4
0.2
0.0
1 0 14 17 13
Class

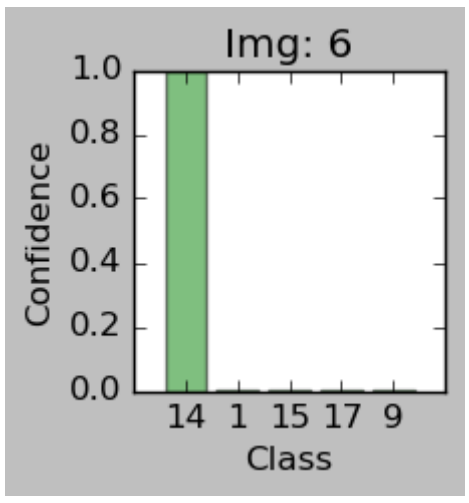

Img: 1
Confidence
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1 14 17 0 25
Class


Img: 2
Confidence
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
14 3 10 9 17
Class


Img: 3
Confidence
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
14 17 1 25 22
Class


Img: 4
Confidence
1.0
0.8
0.6
0.4
0.2
0.0
14 22 15 25 1
Class




## Question 8

*Use the model's softmax probabilities to visualize the **certainty** of its predictions, `tf.nn.top_k` (https://www.tensorflow.org/versions/r0.11/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)*

**Answer:**

The correct label, 14 (stop sign), always appears in the top 5 predictions.
Unfortunately with Image 0 and 5, the model is extremely certain of incorrect labels.
Image 1 is predicted incorrectly, but it is less certain, and somewhat correctly leaning toward second guess of correct answer.
The remaining images: 2,3,4,6 are all labelled as the correct answer with good confidence.

## Question 9

*If necessary, provide documentation for how an interface was built for your model to load and classify newly-acquired images.*

**Answer:**

I used the same exact graph, without dropout, wrapped it in a function, and instead of using random init weights I loaded them from my saved numpy arrays. I cropped the images by hand, scp'ed them to my linux box, loaded the images in RGB shape 32,32,3 numpy arrays, ran them through the same preprocessing functions, and passed them to my same TestArray function.

> **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.