Self-Driving Car Engineer Nanodegree

Deep Learning

Project: Build a Traffic Sign Recognition Classifier

For this project, we can see each of the following setups as outlined below, for necessary completion of the rubric points. The next 5 sections expound on these details.

1. Load the data set
2. Explore, summarize and visualize the data set
3. Design, train and test a model architecture
4. Use the model to make predictions on new images
5. Analyze the softmax probabilities of the new images
6. Summarize the results with a written report

   This here Jupyter notebook is the written report!

Step 0: All needed imports

Useful for testing if environment is ready to rip

In [1]:
```
import csv
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import pickle
import random

from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow.contrib.layers import flatten

print('Loaded all imports!')
```

```
Loaded all imports!
```

Step 1: Load the data set

Retreived file via curl piping into a zip file and extracting. Per project description, [dataset may be found at the provided link (https://d17h27t6h515a5.cloudfront.net/topher/2017/February/5898cd6f_traffic-signs-data/traffic-signs-data.zip)](https://d17h27t6h515a5.cloudfront.net/topher/2017/February/5898cd6f_traffic-signs-data/traffic-signs-data.zip)

In [2]:
```
# Load pickled data

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

# sign names for labels
names_file      = './signnames.csv'

with open(names_file) as f:
    reader = csv.reader(f)
    next(reader) # skip first row
    names = {int(rows[0]):rows[1] for rows in reader}

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']

print('all done loading')
```

```
all done loading
```

Step 2: Dataset Summary & Exploration

The next few section provide a snapshot of the data and visualizes some examples.

Here is Basic Summary of the Data Set

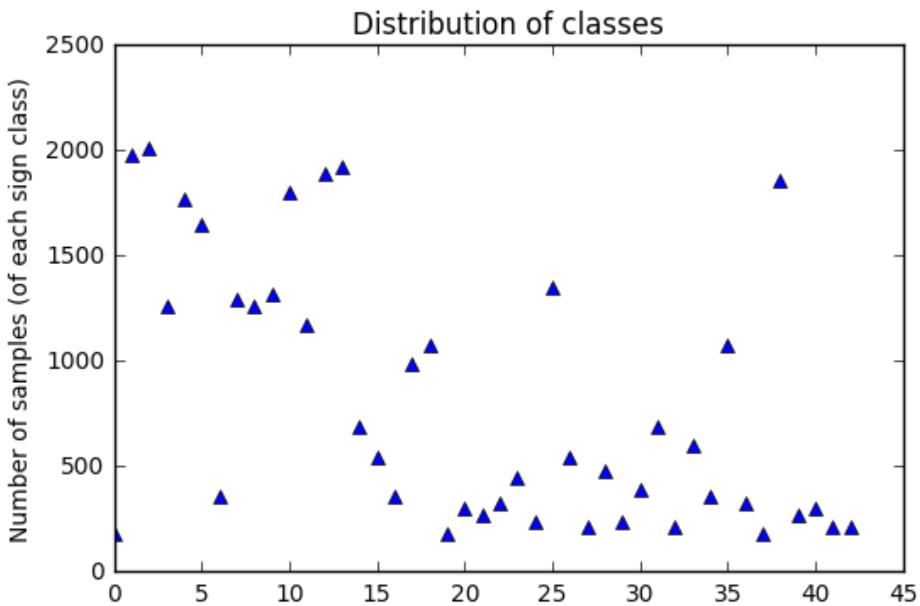
In [3]:
```
# Number of training examples
n_train = np.shape(X_train)[0]

# Number of validation examples
n_validation = np.shape(X_valid)[0]

# Number of testing examples.
n_test = np.shape(X_test)[0]

# What's the shape of an traffic sign image?
image_shape = np.shape(X_train[0])

# How many unique classes/labels there are in the dataset.
sign_classes, class_counts = np.unique(y_train, return_counts = True)
num_classes = len(set(y_train)) # Original way I came up with, found this way abo
ve when needing to do augmentation

print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", len(sign_classes))
```

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

here is an exploratory visualization of the dataset

As can be clearly seen, the distribution of classes if very uneven

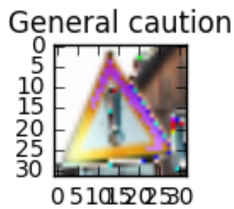
In [88]:
```
%matplotlib inline

plt.title("Distribution of classes")
plt.ylabel('Number of samples (of each sign class)')
chart = plt.plot(class_counts, 'b^')
#chart.set_xticklabels(names) #TODO Actually name the signs on the bar chart so w
e know what is underrepresented\
#plt.bar(x, y, width, color="blue")
plt.show()
```



...and finally, a sampling of some images from the dataset

I ran this after shuffling and normalizing the dataset. Hence no need to randomize index. Commented in case needed

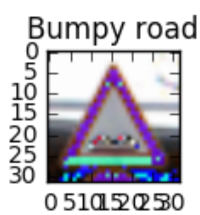
In [114]:
```
%matplotlib inline

for x in range(5):
    index = x #random.randint(0, len(X_train))
    image = X_train[index].squeeze()

    plt.figure(figsize=(1,1))
    plt.title(names[y_train[index]])
    #plt.subplot(1,10,x+1) # failed attempt to inline some of the images, not ger
mane to this work
    plt.imshow(image) # showing with cmap='gray' does not have effect; per docs,
if array depth is 3, then cmap is ignored
    plt.show()
    print(y_train[x])
```



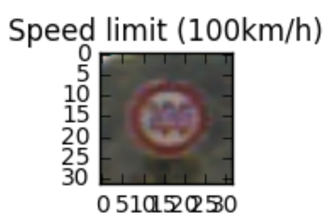
18

38

15



22



7

Step 3: Design, train and test a model architecture

Here we setup the model architecture, pre-process the data and other fun things.

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

In [4]:
```
X_train, y_train = shuffle(X_train, y_train)
X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train,
test_size=0.2, random_state=0)

print('shuffled up and training/validation sets ready')
```

```
shuffled up and training/validation sets ready
```

In [ ]:
```
def grayscale(img):
    """Applies the Grayscale transform
    This will return an image with only one color channel
    but NOTE: to see the returned image as grayscale
    (assuming your grayscaled image is called 'gray')
    you should call plt.imshow(gray, cmap='gray')"""
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

new_X_train = []
for idx,x in enumerate(X_train):
    new_X_train.append(reshape(grayscale(x), (32, 32)))
X_train = new_X_train

new_X_validation = []
for idx,x in enumerate(X_validation):
    new_X_validation.append(reshape(grayscale(x), (32, 32)))
X_validation = new_X_validation

new_X_test = []
for idx,x in enumerate(X_test):
    new_X_test.append(reshape(grayscale(x), (32, 32)))
X_test = new_X_test

print('Grayscaled all images')
```

In [ ]:
```
def rough_normalization(pixels):
    return (pixels - 128) / 128

clahe = cv2.createCLAHE()
def clahe_normalize(image):
    return clahe.apply(image)

# Kudos to https://stackoverflow.com/a/28520445
def image_histogram_equalization(image, number_bins=256):
#def normalize(pixels):
    # from http://www.janeriksolem.net/2009/06/histogram-equalization-with-python
-and.html

    # get image histogram
    image_histogram, bins = np.histogram(image.flatten(), number_bins, normed=Tru
e)
    cdf = image_histogram.cumsum() # cumulative distribution function
    cdf = 255 * cdf / cdf[-1] # normalize

    # use linear interpolation of cdf to find new pixel values
    image_equalized = np.interp(image.flatten(), bins[:-1], cdf)

    return image_equalized.reshape(image.shape)#, cdf

normalize = image_histogram_equalization # choose normalization approach here

for idx,x in enumerate(X_train):
    X_train[idx] = normalize(x)

for idx,x in enumerate(X_validation):
    X_validation[idx] = normalize(x)

for idx,x in enumerate(X_test):
    X_test[idx] = normalize(x)

print('Images are now normalized via {}, hooray!'.format(normalize))
```

Model Architecture

Description

I feel the code below provides some of the relevant hyper-parameters used for tuning and the deep network architecture. This is basically a copy of the LeNet solution, with DropOut and regularization added being the main difference. In addition, one of the fully-connected layers was dropped. The other main difference is the depths for the convolution layers were upped a bit more as the LeNet solution was for 10 classes, and since we have 43, there are surely more finer-grained features to learn.

In [5]:
```
EPOCHS = 1500
BATCH_SIZE = 2048 #orig 128
# Arguments used for tf.truncated_normal, randomly defines variables for the weig
hts and biases for each layer
mu = 0
sigma = 0.18

# Declaring weights outside method for access later.  CLEANME: Move this to a pro
per class
depth_conv1 = 30
depth_conv2 = 60
depth_conv3 = 120
size_fc_1 = 120
size_fc_2 = 45

conv1_W = tf.Variable(tf.truncated_normal(shape=(3, 3, 3, depth_conv1), mean = mu
, stddev = sigma))
conv2_W = tf.Variable(tf.truncated_normal(shape=(4, 4, depth_conv1, depth_conv2),
mean = mu, stddev = sigma))
conv3_W = tf.Variable(tf.truncated_normal(shape=(3, 3, depth_conv2, depth_conv3),
mean = mu, stddev = sigma))
fc1_W = tf.Variable(tf.truncated_normal(shape=(36*depth_conv2, size_fc_1), mean =
mu, stddev = sigma))
fc2_W  = tf.Variable(tf.truncated_normal(shape=(size_fc_1, size_fc_2), mean = mu,
stddev = sigma))
fc3_W  = tf.Variable(tf.truncated_normal(shape=(size_fc_2, num_classes), mean = m
u, stddev = sigma))
keep_prob = tf.placeholder(tf.float32) # probability to keep units

# from gentle guide to batch
phase = tf.placeholder(tf.bool, name='phase')

def LeNet(x):
    print("Charging up!")

    #print(x)
    # Layer 1: Convolutional. Input = 32x32x3. Output = 28x28xdepth_conv1.
    conv1_b = tf.Variable(tf.zeros(depth_conv1))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + c
onv1_b

    # Activation.
    conv1 = tf.nn.relu(conv1)
    #sess.run(tf.shape(conv1))
    #conv1 =  tf.nn.dropout(conv1, keep_prob)

    # Pooling. Input = 28x28xdepth_conv1. Output = 14x14xdepth_conv1.
     # where 28 = 32 - 4, where 32 is original size, and '4' is size of convoluti
onal filter..
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddi
ng='VALID')

    # Layer 2: Convolutional. Output = 10x10xdepth_conv2.
    conv2_b = tf.Variable(tf.zeros(depth_conv2))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID')
+ conv2_b

    # Activation.
    conv2 = tf.nn.relu(conv2)
    #conv2 = tf.nn.dropout(conv2, keep_prob)

    # Pooling. Input = 10x10xdepth_conv2. Output = 5x5xdepth_conv2.
```

```
All done
```

Train, Validate and Test the Model

In [21]:
```
%%time

rate = 0.000675
largeBiasPenalty = 0.065
keep_probability = 0.72

logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=
logits)

# Add regularization to the loss for highly-biased weights
loss_operation = tf.reduce_mean(cross_entropy) + \
    largeBiasPenalty * tf.nn.l2_loss(conv1_W) +\
    largeBiasPenalty * tf.nn.l2_loss(conv2_W) +\
    largeBiasPenalty * tf.nn.l2_loss(fc1_W) +\
    largeBiasPenalty * tf.nn.l2_loss(fc2_W) +\
    largeBiasPenalty * tf.nn.l2_loss(fc3_W)

optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)


### Calculate and report the accuracy on the training and validation set.
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset
+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y
, keep_prob: 1.0, phase: False})
        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):
        print("EPOCH {} ...".format(i+1))
        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, keep_
prob: keep_probability, phase: True})

        training_accuracy = evaluate(X_train, y_train)
        validation_accuracy = evaluate(X_validation, y_validation)
        print("{:.3f} = Training accuracy vs Validation Accuracy = {:.3f}".format
(training_accuracy, validation_accuracy))

        print("Starting to save model...")
```

```
Charging up!
Training...

EPOCH 1 ...
0.065 = Training accuracy vs Validation Accuracy = 0.061
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 2 ...
0.055 = Training accuracy vs Validation Accuracy = 0.054
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 3 ...
0.070 = Training accuracy vs Validation Accuracy = 0.068
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 4 ...
0.080 = Training accuracy vs Validation Accuracy = 0.077
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 5 ...
0.127 = Training accuracy vs Validation Accuracy = 0.120
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 6 ...
0.144 = Training accuracy vs Validation Accuracy = 0.139
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 7 ...
0.172 = Training accuracy vs Validation Accuracy = 0.169
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 8 ...
0.218 = Training accuracy vs Validation Accuracy = 0.210
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 9 ...
0.293 = Training accuracy vs Validation Accuracy = 0.283
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 10 ...
0.380 = Training accuracy vs Validation Accuracy = 0.372
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 11 ...
0.437 = Training accuracy vs Validation Accuracy = 0.425
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 12 ...
0.480 = Training accuracy vs Validation Accuracy = 0.474
Starting to save model...
...model saved.  Onward, hiya!
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<timed exec> in <module>()

<timed exec> in evaluate(X_data, y_data)

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in run(self, fetches, feed_dict, options, run_metadata)
    764     try:
    765       result = self._run(None, fetches, feed_dict, options_ptr,
--> 766                          run_metadata_ptr)
    767       if run_metadata:
    768         proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _run(self, handle, fetches, feed_dict, options, run_me
tadata)
    962     if final_fetches or final_targets:
    963       results = self._do_run(handle, final_targets, final_fetches,
--> 964                              feed_dict_string, options, run_metadata)
    965     else:
    966       results = []

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _do_run(self, handle, target_list, fetch_list, feed_di
ct, options, run_metadata)
   1012     if handle is None:
   1013       return self._do_call(_run_fn, self._session, feed_dict, fetch_list
,
-> 1014                            target_list, options, run_metadata)
   1015     else:
   1016       return self._do_call(_prun_fn, self._session, handle, feed_dict,

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _do_call(self, fn, *args)
   1019   def _do_call(self, fn, *args):
   1020     try:
-> 1021       return fn(*args)
   1022     except errors.OpError as e:
   1023       message = compat.as_text(e.message)

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _run_fn(session, feed_dict, fetch_list, target_list, o
ptions, run_metadata)
   1001         return tf_session.TF_Run(session, options,
   1002                                  feed_dict, fetch_list, target_list,
-> 1003                                  status, run_metadata)
   1004
   1005     def _prun_fn(session, handle, feed_dict, fetch_list):

KeyboardInterrupt:
```

And what do we get!!?? PASS!!!! Kinda...

In [22]:
```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

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

```
Test Accuracy = 0.882
```

Below preserves the one time I actually got the required accuracy. Yes, its dependent on how the dataset is shuffled, but I have spent too long on this project, need to catch up with the rest of the class!

In [133]:
```
%%time

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

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

```
Test Accuracy = 0.932
CPU times: user 16.3 s, sys: 3.6 s, total: 19.9 s
Wall time: 15.4 s
```

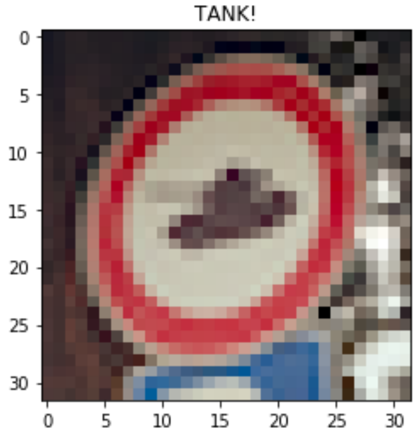
Step 4: Use the model to make predictions on new images

Here we download 5 images from the web to test our dataset against. I added a sixth image. For some silly reason when I first started, I sought out traffic signs that are *not* in the dataset (we, afterall, have a test set for verifying overall accuracy anyway). So I included a sixth one, a Tank-cross sign, which doesn't exist in the dataset, to see how easily the model may be fooled.

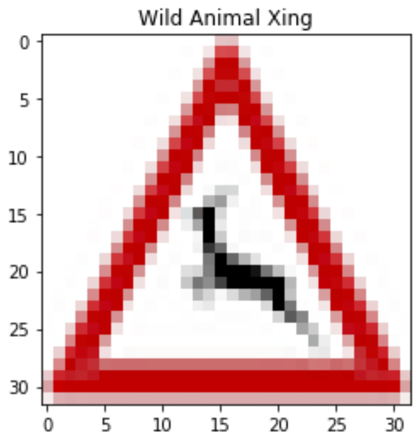
Load and Output the Images

In [27]:
```
# class id = NONE!
tank = mpimg.imread('downloaded/tank.png')
print('size of tank: {} '.format(np.shape(tank)))

# It seems like mpimg is already 'normalizing' the images as we read them.  Or it
was OSX Preview as I reshapped the classes.
#print('full output of tank: {}'.format(tank))
#print('normalizing tank: {}'.format(normalize(tank)))

plt.title('TANK!')
plt.imshow(tank)
plt.show()

# class id = 31
wild_animals = mpimg.imread('downloaded/847px-Zeichen_142-10_-_Wildwechsel,_Aufst
ellung_rechts,_StVO_1992.svg.png')
print('size of animal: {} '.format(np.shape(wild_animals)))
plt.title('Wild Animal Xing')
plt.imshow(wild_animals)
plt.show()

# class id = 18
general_caution = mpimg.imread('downloaded/Zeichen_101_-_Gefahrstelle,_StVO_1970.
svg.png')
print('size of general_caution: {} '.format(np.shape(general_caution)))
plt.title('General Caution')
plt.imshow(general_caution)
plt.show()

# class id = 19
dangerous_left = mpimg.imread('downloaded/Zeichen_103-10_-_Kurve_(links),_StVO_19
92.svg.png')
print('size of dangerous_left: {} '.format(np.shape(dangerous_left)))
plt.title('Dangerous curve to the left')
plt.imshow(dangerous_left)
plt.show()


# class id = 3
kph_60 = mpimg.imread('downloaded/Zeichen_274-56.svg.png')
print('size of kph_60: {} '.format(np.shape(kph_60)))
plt.title('60 KM/h')
plt.imshow(kph_60)
plt.show()

# class id 1
kph_30 = mpimg.imread('downloaded/Zeichen_274.1_-_Beginn_einer_Tempo_30-Zone,_StV
O_2013.svg.png')
print('size of kph_30: {} '.format(np.shape(kph_30)))
plt.title('Entering 30 KM/h zone')
plt.imshow(kph_30)
plt.show()



X_downloaded = [tank, wild_animals, general_caution, dangerous_left, kph_60, kph_
30]
y_downloaded = [-1,31,18,19,3,1]

print(np.shape(X_downloaded))
```

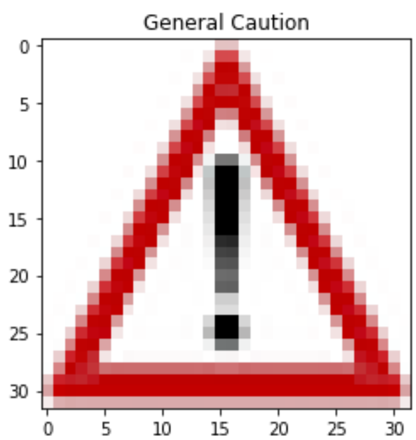
```
size of tank: (32, 32, 3)
```



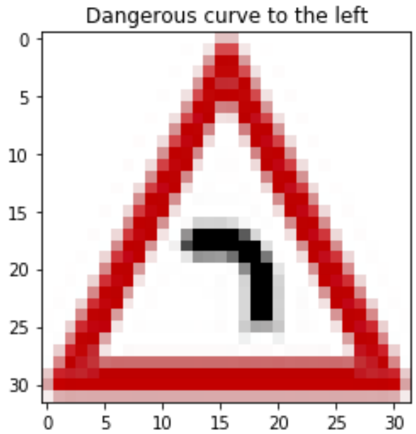

```
size of animal: (32, 32, 3)
```



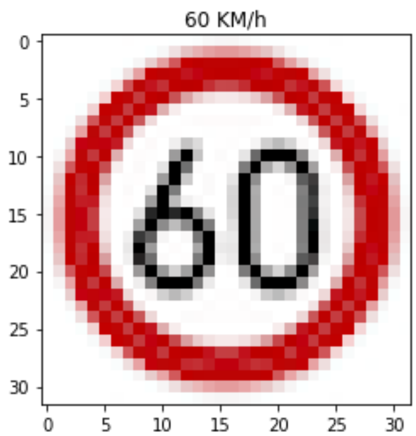

```
size of general_caution: (32, 32, 3)
```



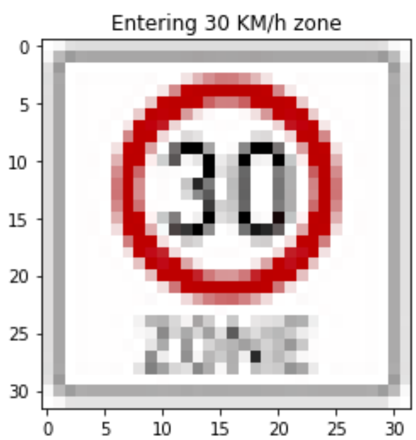

```
size of dangerous_left: (32, 32, 3)
```




```
size of kph_60: (32, 32, 3)
```




```
size of kph_30: (32, 32, 3)
```




```
(6, 32, 32, 3)
```

Step 5 Analyze the softmax probabilities of the new images Test a Model on New Images

Predict the Sign Type for Each Image

In [23]:
```
%%time

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    test_accuracy = evaluate(X_downloaded, y_downloaded)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Test Accuracy = 0.000
CPU times: user 564 ms, sys: 460 ms, total: 1.02 s
Wall time: 528 ms
```

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

In [24]:
```
# function to print out softmax
# It may be a bit more optimized to run this during accuracy option. But not need
ed for this project's purposes

top5 = tf.nn.top_k(tf.nn.softmax(logits), k=5)

def eval_downloaded(X_data, y_data):
    num_examples = len(X_data)
    sess = tf.get_default_session()
    t5 = sess.run(top5, feed_dict={x: X_data, y: y_data, keep_prob: 1.0})
    print("TOP5   {}".format(t5))
    return t5

print('Eval with top5 function loaded')
```

```
Eval with top5 function loaded
```

In [25]:
```
%%time

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    top5_accuracy = eval_downloaded(X_downloaded, y_downloaded)
    print("top 5 matrix in full = {}".format(top5_accuracy))
```

```
TOP5   TopKV2(values=array([[ 0.0981414 ,  0.07578082,  0.06946859,  0.06201295,
0.04533249],
       [ 0.09399547,  0.08496545,  0.06298173,  0.06221561,  0.04303629],
       [ 0.08181851,  0.07178923,  0.07077026,  0.06547637,  0.04611111],
       [ 0.10083228,  0.08288508,  0.07090969,  0.05555868,  0.04724773],
       [ 0.09955632,  0.08115049,  0.08067164,  0.06269072,  0.04845662],
       [ 0.08280152,  0.07337368,  0.07168196,  0.0580777 ,  0.04537925]], dtype
=float32), indices=array([[10,  5,  3,  8,  9],
       [10,  5,  8,  3, 25],
       [ 5,  8, 10,  3, 20],
       [10,  5,  3,  8, 20],
       [ 5,  3, 10,  8,  4],
       [10,  5,  3,  8, 20]], dtype=int32))
top 5 matrix in full = TopKV2(values=array([[ 0.0981414 ,  0.07578082,  0.069468
59,  0.06201295,  0.04533249],
       [ 0.09399547,  0.08496545,  0.06298173,  0.06221561,  0.04303629],
       [ 0.08181851,  0.07178923,  0.07077026,  0.06547637,  0.04611111],
       [ 0.10083228,  0.08288508,  0.07090969,  0.05555868,  0.04724773],
       [ 0.09955632,  0.08115049,  0.08067164,  0.06269072,  0.04845662],
       [ 0.08280152,  0.07337368,  0.07168196,  0.0580777 ,  0.04537925]], dtype
=float32), indices=array([[10,  5,  3,  8,  9],
       [10,  5,  8,  3, 25],
       [ 5,  8, 10,  3, 20],
       [10,  5,  3,  8, 20],
       [ 5,  3, 10,  8,  4],
       [10,  5,  3,  8, 20]], dtype=int32))
CPU times: user 564 ms, sys: 516 ms, total: 1.08 s
Wall time: 545 ms
```

Overall, these numbers are not surprising. Whatever preprocessing steps I did on the images clearly did not go over so well. Given more time (must move on to next project), I am confident I can get this to be more interesting. At least I can show that I know enough Tensorflow to evaluate top-5. What would be interesting would be one would expect to hope to see low probabilities for a completely unknown image. This would maybe also require more augmentation.

Step 6 Project Writeup

This here Jupyter notebook acts as my writeup!

Summary Overview

Overall, I had a frustrating time on this project, as fun as it was to play around with tensorflow and learn actual traffic signs. Due to work obligations, I could realisitically only commit an hour or two per day to this project. The majority of my time I spent spinning my wheels due to trying to get the saved model to restore from the saved checkpoint. Using 'last checkpoint' works fine, but the other approaches as documented in the TensorFlow documentation and Udacity class only resulted in errors.

I also was surprised that the LeNet architecture worked prefectly fine, even without normalization and in spite of the fact it had been setup for 10-class problem, rather than the 43-class problem we currently have. Also surprising is adding dropout dramatically decreased the accuracy. I noticed that, in spite of a lot of parameters, there seemed to be a local optima that kept accuracy stuck at exactly 5.4%. Sometimes, I would let the model run for 70 epochs and then it would rapidly get close to 95% accuracy after getting out of the 5.4% rut it was in.

I do not agree with augmenting the dataset, despite explicit hints provided, and feel it's important to construct a deep net that realisitically simulates learning conditions of the real world. Even better would be to use few examples. The average human, in their lifetime, would not need to see 100,000+ examples of traffic signs repeatedly in order to accurately predict them. Granted, we are also in these training sets teaching a net to read images. It would be nice if there were preconfigured nets that already understand how to read images, that then feed into one specifically for traffic signs. Some of my comments are inspired by a tweet from Yan Lecunn, I'll let it speak for itself when he was congratulating Alpha Go on its achievements, and then challenged them to do the same with far less examples and reinforcement learning. Yann's tweet (https://twitter.com/ylecun/status/708732919548919808)

In either case I was able to get the minimun necessary accuracy at least once, as saved above.

Preprocessing Steps Taken

I applied a grand-total of two pre-processing steps. This was inspired by Yann Lecunn's paper about how they did not bother to apply HOG or other typical image transformations, preferring to allow the network to learn it's own.

The two I took are

- Shuffling the test set and leaving out a few for validation
- Applying image histogram equalization, which came from a stackoverflow question. This works for the multi-color images that I choose to keep instead of going with grayscale

I also experimented, as suggested by reviewer, with CLAHE normalization, and found it only worked for grayscale and didn't seem to improve performance.

Questions to specifically respond to in writeup and in response to feedback

What was the first architecture I tried and why was it chosen?

I went with LeNet since that was the easiest starting part. I also had a rough time getting the model restoration to work as I apparently kept mixing up the multiple versions of TensorFlow out there, so answers to some of the errors I encountered were not easy to come by. In addition, it provided good out-of-the-box performance.

Problems with initial architecture?

In [ ]:

```
```