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)

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

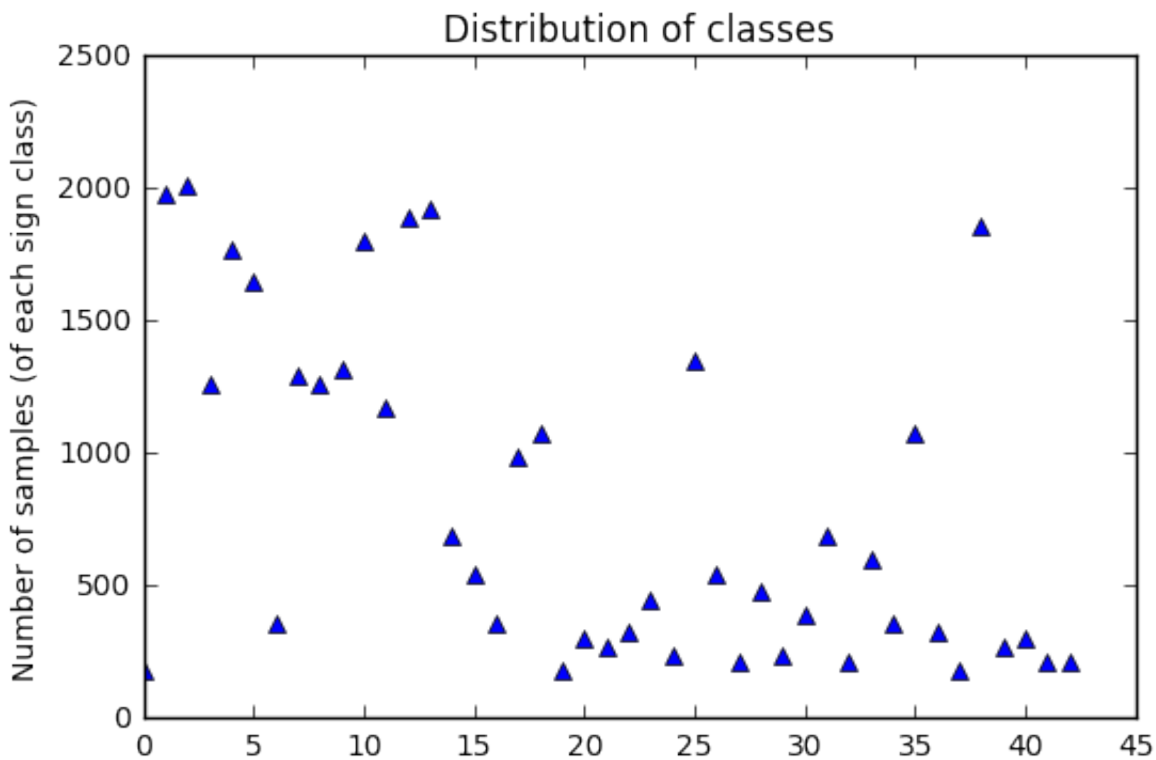
In [3]:
```
# Number of training examples
n_train = np.shape(X_train)[0]

# 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 t
his way above 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

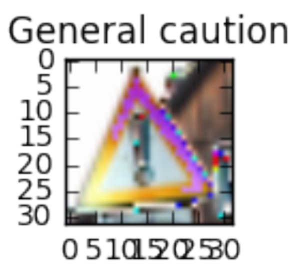
In [88]:
```
%matplotlib inline

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 we 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 imag
es, not germane 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])
```

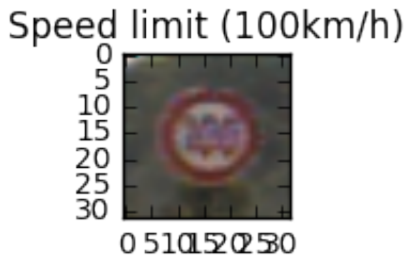


18

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_trai
n, 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 [5]:
```
def normalize(pixels):
    return (pixels - 128) / 128

X_train = normalize(X_train)
X_validation = normalize(X_validation)
X_test = normalize(X_test)

print('Images are now roughly normalized, hooray!')
```

```
Images are now roughly normalized, hooray!
```

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 = 500
BATCH_SIZE = 1024 #orig 128
# Arguments used for tf.truncated_normal, randomly defines variables f
or the weights and biases for each layer
mu = 0
sigma = 0.16

# Declaring weights outside method for access later.  CLEANME: Move th
is to a proper class
depth_conv1 = 6
depth_conv2 = 16
size_fc_1 = 120
size_fc_2 = 84

conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, depth_conv1)
, mean = mu, stddev = sigma))
conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, depth_conv1, de
pth_conv2), mean = mu, stddev = sigma))
fc1_W = tf.Variable(tf.truncated_normal(shape=(5*5*depth_conv2, size_f
c_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 = mu, stddev = sigma))
keep_prob = tf.placeholder(tf.float32) # probability to keep units
```

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

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28xdepth_co
nv1.
    conv1_b = tf.Variable(tf.zeros(depth_conv1))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='
VALID') + conv1_b

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

    # Pooling. Input = 28x28xdepth_conv1. Output = 14x14xdepth_conv1.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2
, 1], padding='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], paddi
ng='VALID') + conv2_b

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

    # Pooling. Input = 10x10xdepth_conv2. Output = 5x5xdepth_conv2.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2
, 1], padding='VALID')

    # Flatten. Input = 5x5xdepth_conv2. Output = 1800.
    fc0   = flatten(conv2)

    # Layer 3: Fully Connected. Input = 1800. Output = size_fc_1.
    fc1_b = tf.Variable(tf.zeros(size_fc_1))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

    # Activation.
    fc1    = tf.nn.relu(fc1)
    #fc1    = tf.nn.dropout(fc1, keep_prob)

    # Layer 3: Fully Connected. Input = size_fc_1. Output = size_fc_2.
    fc2_b  = tf.Variable(tf.zeros(size_fc_2))
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

    # Activation.
    fc2    = tf.nn.relu(fc2)
    fc2    = tf.nn.dropout(fc2, keep_prob)
```

```
    # Layer 4: Fully Connected. Input = size_fc_2. Output = num_classe
s.
    fc3_b  = tf.Variable(tf.zeros(num_classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits

x = tf.placeholder(tf.float32, (None, 32, 32, 3))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, num_classes)

print('All done')
```

```
All done
```

Train, Validate and Test the Model

In [21]:

```
%%time

rate = 0.00175
largeBiasPenalty = 0.025
keep_probability = 0.79

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 s
et.
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_
y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.flo
at32))
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[of
fset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x,
y: batch_y, keep_prob: 1.0})
        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: bat
ch_y, keep_prob: keep_probability})

        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...")
        saver.save(sess, './traffick')
        print("...model saved.  Onward, hiya!")

        print()

print("ALL DONE!  Yippie Kay Yae!")
```

```
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!

EPOCH 13 ...
0.520 = Training accuracy vs Validation Accuracy = 0.511
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 14 ...
0.532 = Training accuracy vs Validation Accuracy = 0.520
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 15 ...
0.562 = Training accuracy vs Validation Accuracy = 0.549
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 16 ...
0.562 = Training accuracy vs Validation Accuracy = 0.549
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 17 ...
0.591 = Training accuracy vs Validation Accuracy = 0.578
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 18 ...
0.614 = Training accuracy vs Validation Accuracy = 0.598
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 19 ...
0.649 = Training accuracy vs Validation Accuracy = 0.637
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 20 ...
0.666 = Training accuracy vs Validation Accuracy = 0.654
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 21 ...
0.701 = Training accuracy vs Validation Accuracy = 0.693
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 22 ...
0.701 = Training accuracy vs Validation Accuracy = 0.690
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 23 ...
0.732 = Training accuracy vs Validation Accuracy = 0.718
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 24 ...
0.737 = Training accuracy vs Validation Accuracy = 0.718
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 25 ...
0.756 = Training accuracy vs Validation Accuracy = 0.744
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 26 ...
0.781 = Training accuracy vs Validation Accuracy = 0.765
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 27 ...
0.795 = Training accuracy vs Validation Accuracy = 0.783
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 28 ...
0.795 = Training accuracy vs Validation Accuracy = 0.787
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 29 ...
0.803 = Training accuracy vs Validation Accuracy = 0.790
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 30 ...
0.805 = Training accuracy vs Validation Accuracy = 0.790
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 31 ...
0.830 = Training accuracy vs Validation Accuracy = 0.816
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 32 ...
0.823 = Training accuracy vs Validation Accuracy = 0.810
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 33 ...
0.830 = Training accuracy vs Validation Accuracy = 0.811
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 34 ...
0.843 = Training accuracy vs Validation Accuracy = 0.820
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 35 ...
0.854 = Training accuracy vs Validation Accuracy = 0.835
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 36 ...
0.843 = Training accuracy vs Validation Accuracy = 0.823
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 37 ...
0.861 = Training accuracy vs Validation Accuracy = 0.840
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 38 ...
0.872 = Training accuracy vs Validation Accuracy = 0.853
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 39 ...
0.873 = Training accuracy vs Validation Accuracy = 0.856
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 40 ...
0.874 = Training accuracy vs Validation Accuracy = 0.855
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 41 ...
0.880 = Training accuracy vs Validation Accuracy = 0.859
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 42 ...
0.884 = Training accuracy vs Validation Accuracy = 0.868
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 43 ...
0.875 = Training accuracy vs Validation Accuracy = 0.854
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 44 ...
0.897 = Training accuracy vs Validation Accuracy = 0.877
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 45 ...
0.898 = Training accuracy vs Validation Accuracy = 0.879
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 46 ...
0.910 = Training accuracy vs Validation Accuracy = 0.894
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 47 ...
0.901 = Training accuracy vs Validation Accuracy = 0.888
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 48 ...
0.910 = Training accuracy vs Validation Accuracy = 0.892
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 49 ...
0.918 = Training accuracy vs Validation Accuracy = 0.903
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 50 ...
0.918 = Training accuracy vs Validation Accuracy = 0.903
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 51 ...
0.929 = Training accuracy vs Validation Accuracy = 0.914
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 52 ...
0.924 = Training accuracy vs Validation Accuracy = 0.903
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 53 ...
0.935 = Training accuracy vs Validation Accuracy = 0.915
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 54 ...
0.935 = Training accuracy vs Validation Accuracy = 0.915
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 55 ...
0.937 = Training accuracy vs Validation Accuracy = 0.918
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 56 ...
0.941 = Training accuracy vs Validation Accuracy = 0.919
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 57 ...
0.942 = Training accuracy vs Validation Accuracy = 0.921
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 58 ...
0.943 = Training accuracy vs Validation Accuracy = 0.924
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 59 ...
0.941 = Training accuracy vs Validation Accuracy = 0.923
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 60 ...
0.946 = Training accuracy vs Validation Accuracy = 0.924
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 61 ...
0.938 = Training accuracy vs Validation Accuracy = 0.922
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 62 ...
0.953 = Training accuracy vs Validation Accuracy = 0.934
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 63 ...
0.947 = Training accuracy vs Validation Accuracy = 0.929
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 64 ...
0.949 = Training accuracy vs Validation Accuracy = 0.928
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 65 ...
0.958 = Training accuracy vs Validation Accuracy = 0.936
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 66 ...
0.956 = Training accuracy vs Validation Accuracy = 0.935
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 67 ...
0.961 = Training accuracy vs Validation Accuracy = 0.938
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 68 ...
0.954 = Training accuracy vs Validation Accuracy = 0.931
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 69 ...
0.963 = Training accuracy vs Validation Accuracy = 0.940
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 70 ...
0.959 = Training accuracy vs Validation Accuracy = 0.939
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 71 ...
0.962 = Training accuracy vs Validation Accuracy = 0.938
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 72 ...
0.962 = Training accuracy vs Validation Accuracy = 0.942
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 73 ...
0.963 = Training accuracy vs Validation Accuracy = 0.943
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 74 ...
0.966 = Training accuracy vs Validation Accuracy = 0.947
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 75 ...
0.964 = Training accuracy vs Validation Accuracy = 0.941
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 76 ...
0.955 = Training accuracy vs Validation Accuracy = 0.933
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 77 ...
0.960 = Training accuracy vs Validation Accuracy = 0.942
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 78 ...
0.972 = Training accuracy vs Validation Accuracy = 0.952
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 79 ...
0.966 = Training accuracy vs Validation Accuracy = 0.946
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 80 ...
0.968 = Training accuracy vs Validation Accuracy = 0.948
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 81 ...
0.969 = Training accuracy vs Validation Accuracy = 0.947
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 82 ...
0.970 = Training accuracy vs Validation Accuracy = 0.950
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 83 ...
0.974 = Training accuracy vs Validation Accuracy = 0.955
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 84 ...
0.972 = Training accuracy vs Validation Accuracy = 0.951
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 85 ...
0.971 = Training accuracy vs Validation Accuracy = 0.949
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 86 ...
0.971 = Training accuracy vs Validation Accuracy = 0.951
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 87 ...
0.975 = Training accuracy vs Validation Accuracy = 0.954
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 88 ...
0.966 = Training accuracy vs Validation Accuracy = 0.945
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 89 ...
0.976 = Training accuracy vs Validation Accuracy = 0.959
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 90 ...
0.976 = Training accuracy vs Validation Accuracy = 0.955
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 91 ...
0.977 = Training accuracy vs Validation Accuracy = 0.959
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 92 ...
0.968 = Training accuracy vs Validation Accuracy = 0.944
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 93 ...
0.979 = Training accuracy vs Validation Accuracy = 0.959
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 94 ...
0.975 = Training accuracy vs Validation Accuracy = 0.955
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 95 ...
0.979 = Training accuracy vs Validation Accuracy = 0.960
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 96 ...
0.975 = Training accuracy vs Validation Accuracy = 0.955
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 97 ...
0.980 = Training accuracy vs Validation Accuracy = 0.959
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 98 ...
0.977 = Training accuracy vs Validation Accuracy = 0.958
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 99 ...
0.981 = Training accuracy vs Validation Accuracy = 0.963
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 100 ...
0.977 = Training accuracy vs Validation Accuracy = 0.959
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 101 ...
0.980 = Training accuracy vs Validation Accuracy = 0.961
Starting to save model...
...model saved.  Onward, hiya!
```

```
EPOCH 102 ...
0.980 = Training accuracy vs Validation Accuracy = 0.963
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 103 ...
0.980 = Training accuracy vs Validation Accuracy = 0.962
Starting to save model...
...model saved.  Onward, hiya!

EPOCH 104 ...

---------------------------------------------------------------------
-------
KeyboardInterrupt                         Traceback (most recent cal
l last)
<timed exec> in <module>()

<timed exec> in evaluate(X_data, y_data)

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

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/t
ensorflow/python/client/session.py in _run(self, handle, fetches, fe
ed_dict, options, run_metadata)
    962     if final_fetches or final_targets:
    963       results = self._do_run(handle, final_targets, final_fe
tches,
--> 964                              feed_dict_string, options, run_
metadata)
    965     else:
    966       results = []

/home/carnd/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/t
ensorflow/python/client/session.py in _do_run(self, handle, target_l
ist, fetch_list, feed_dict, 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_metadat
a)
```

```
   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/t
ensorflow/python/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/t
ensorflow/python/client/session.py in _run_fn(session, feed_dict, fe
tch_list, target_list, options, 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

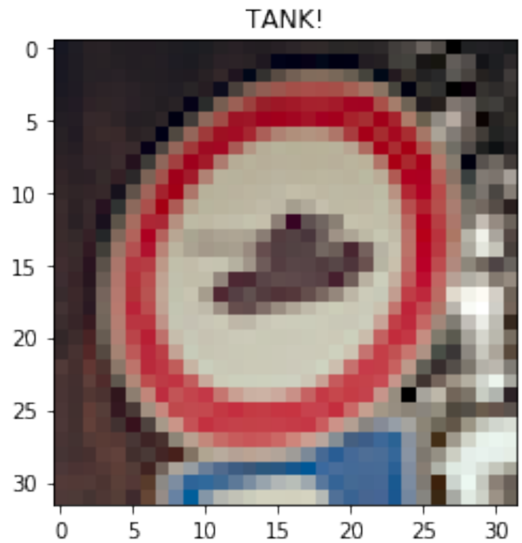
Here we download 5 images from the web to test our dataset against. I added a sixth image. For some silly reason when I first started, I sought out traffic signs that are *not* in the dataset (we, afterall, have a test set for verifying overall accuracy anyway). So I included a sixth one, a Tank-cross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

Load and Output the Images

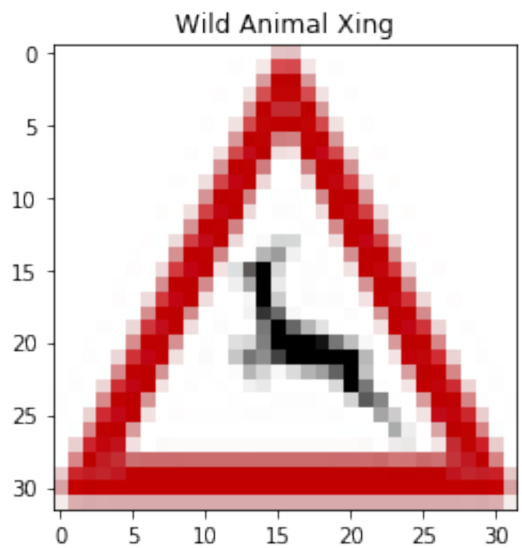
In [27]:
```
# class id = NONE!
tank = mpimg.imread('downloaded/tank.png')
print('size of tank: {} '.format(np.shape(tank)))

# It seems like mpimg is already 'normalizing' the images as we read t
hem.  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_-_Wildwec
hsel,_Aufstellung_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_(link
s),_StVO_1992.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_3
0-Zone,_StVO_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, k
ph_60, kph_30]
y_downloaded = [-1,31,18,19,3,1]

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

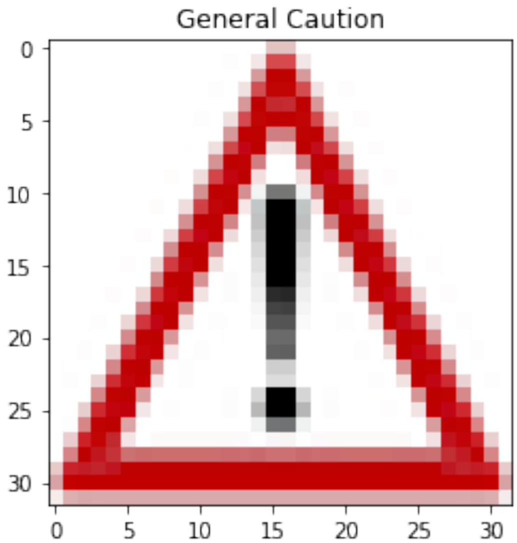
```
size of tank: (32, 32, 3)
```



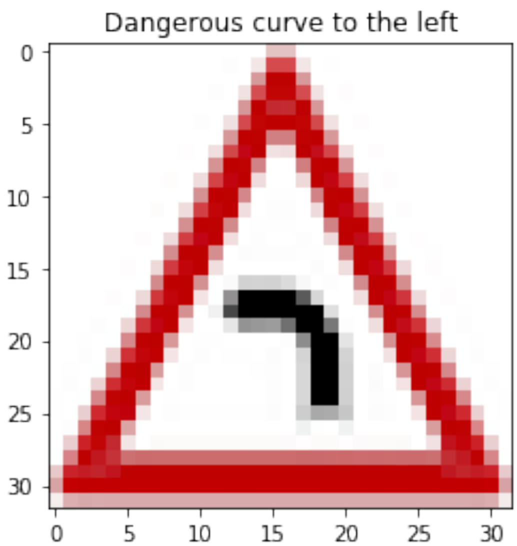
size of animal: (32, 32, 3)



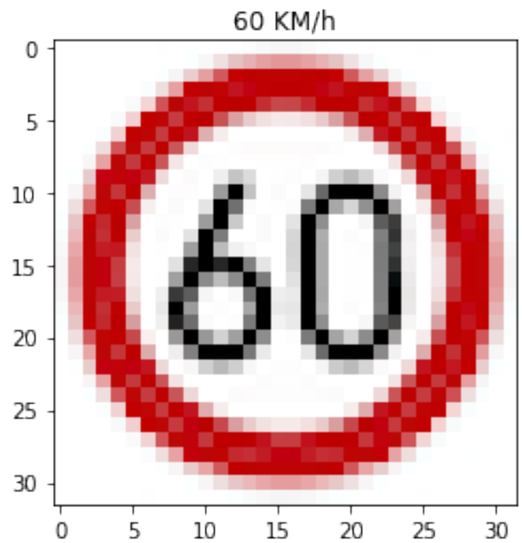
size of general_caution: (32, 32, 3)



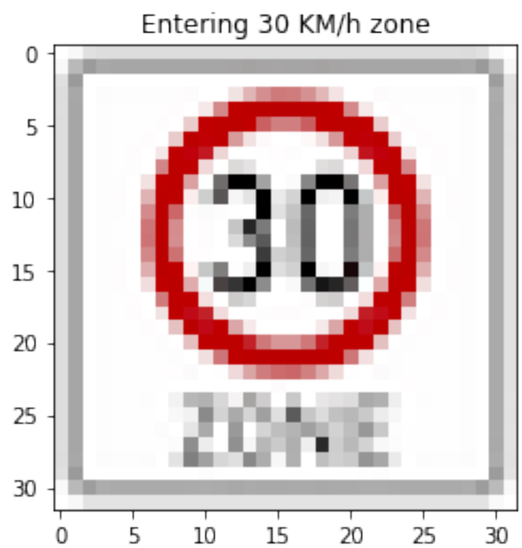
size of dangerous_left: (32, 32, 3)



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. B
ut not needed 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,  0.04303
629],
       [ 0.08181851,  0.07178923,  0.07077026,  0.06547637,  0.04611
111],
       [ 0.10083228,  0.08288508,  0.07090969,  0.05555868,  0.04724
773],
       [ 0.09955632,  0.08115049,  0.08067164,  0.06269072,  0.04845
662],
       [ 0.08280152,  0.07337368,  0.07168196,  0.0580777 ,  0.04537
925]], 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 ,  0.0757808
2,  0.06946859,  0.06201295,  0.04533249],
       [ 0.09399547,  0.08496545,  0.06298173,  0.06221561,  0.04303
629],
       [ 0.08181851,  0.07178923,  0.07077026,  0.06547637,  0.04611
111],
       [ 0.10083228,  0.08288508,  0.07090969,  0.05555868,  0.04724
773],
       [ 0.09955632,  0.08115049,  0.08067164,  0.06269072,  0.04845
662],
       [ 0.08280152,  0.07337368,  0.07168196,  0.0580777 ,  0.04537
925]], 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!

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 rapidally 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.

Some other observations:

- I tried to apply the "keep calm and lower your learning rate" trick. I was giving values as low as a magnitude of 10e-10 and I still constantly kept getting stuck in the 5% rut. I reread what the Adam optimizer is and realize it applies the learn-rate lowering throughout repeated iterations, so I embraced a higher learning rate and that did a much better job of not being stuck in the 5% gutter.
- I removed a fully-connected layer. It took a lot more cajoling to get results above 90%, so I brought it back.
- I used a lazy-person's stop for overregularization. By saving the model after each epoch, I could keep my eye on when it reached a satisfactory accuracy, and stop as needed. I thought about applying a simple-moving average to only stop when the rate of accuracy decreases after X epochs, but since I wasn't sure what values to use, and sometimes it would take dozens of minutes to reach a certain state, I would hate for things to stop. So I used an overly large number of epochs, with manual stopping.
- Strangley, dropout and regularization seemed to make things worst