Self-Driving Car Engineer Nanodegree

Deep Learning

Project: Build a Traffic Sign Recognition Classifier

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

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

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

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

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

Step 0: Load The Data

In [28]:
```
# Load pickled data
import pickle
import os

# TODO: Fill this in based on where you saved the training and testing data

base_path = '/Users/deadman/Google Drive/Udacity Self Driving/Traffic Sign Cla
ssifier/CarND-Traffic-Sign-Classifier-Project/traffic-signs-data'
training_file = os.path.join(base_path, 'train.p')
validation_file= os.path.join(base_path, 'valid.p')
testing_file = os.path.join(base_path, 'test.p')

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

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

Step 1: Dataset Summary & Exploration

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

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

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

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

In [29]:
```
#print(X_train[0].shape)
```

In [30]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results

# TODO: Number of training examples
n_train = len(X_train)

# TODO: Number of validation examples
n_validation = len(X_valid)

# TODO: Number of testing examples.
n_test = len(X_test)

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

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(set(y_train))

print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Number of validation examples = ", n_validation)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

```
Number of training examples = 34799
Number of testing examples = 12630
Number of validation examples =  4410
Image data shape = (32, 32, 3)
Number of classes = 43
```

Include an exploratory visualization of the dataset

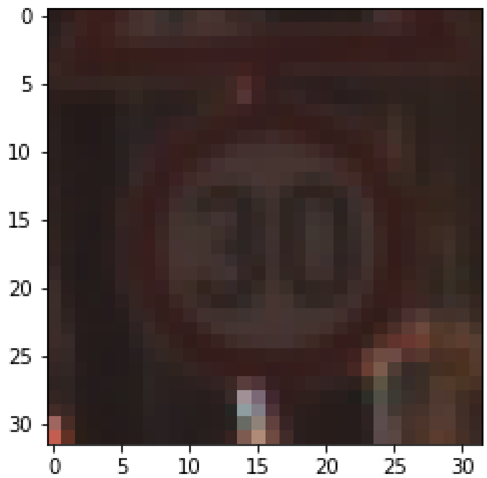
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

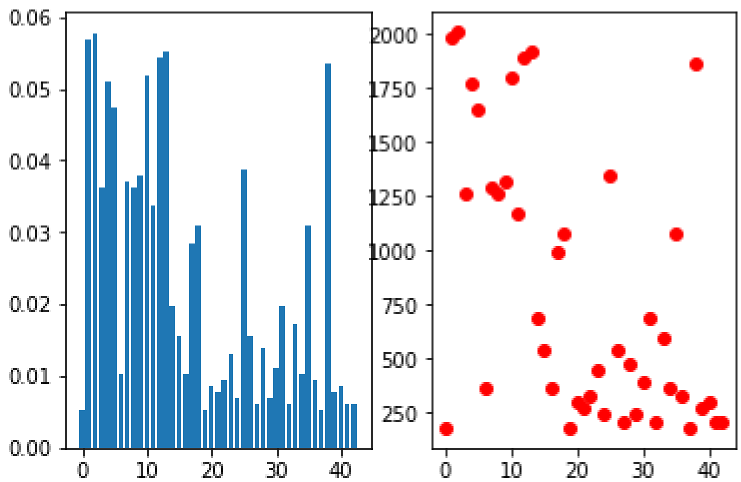
The Matplotlib (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. It can be interesting to look at the distribution of classes in the training, validation and test set. Is the distribution the same? Are there more examples of some classes than others?

In [31]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline
from collections import defaultdict
import numpy as np

# plot a random image
plt.imshow(X_train[2500])

# plot the count of each sign
class_dict = defaultdict(int)
for label in y_train:
    # put in a dictionary
    class_dict[label] += 1

class_freqs = list(class_dict.values())
n_bins = len(class_dict.keys())
print(np.asarray(class_freqs))

fig, ax = plt.subplots(1,2)
ax[1].plot(range(0,n_bins), class_freqs ,'ro')
ax[0].bar(range(0,n_bins), class_freqs/np.sum(class_freqs))
plt.savefig('plots.jpg')
plt.show()
```

```
[ 180 1980 2010 1260 1770 1650  360 1290 1260 1320 1800 1170 1890 1920  690
  540  360  990 1080  180  300  270  330  450  240 1350  540  210  480  240
  390  690  210  599  360 1080  330  180 1860  270  300  210  210]
```





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).

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

With the LeNet-5 solution from the lecture, you should expect a validation set accuracy of about 0.89. To meet specifications, the validation set accuracy will need to be at least 0.93. It is possible to get an even higher accuracy, but 0.93 is the minimum for a successful project submission.

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

- Neural network architecture (is the network over or underfitting?)
- 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.

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

Minimally, the image data should be normalized so that the data has mean zero and equal variance. For image data, `(pixel - 128)/ 128` is a quick way to approximately normalize the data and can be used in this project.

Other pre-processing steps are optional. You can try different techniques to see if it improves performance.

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

In [ ]:

```

```

In [32]:
```
# Pre-processing data here

# convert images to different channel here (lets do grayscale first)
def rgb2gray(img):
    return np.dot(img[...,:3], [0.299, 0.587, 0.114])

# X_train_gray = np.empty(X_train.shape[:3])
# X_valid_gray = np.empty(X_valid.shape[:3])
```

```
# X_test_gray = np.empty(X_test.shape[:3])
# # print(X_train.shape[:3])
# # print(X_train_gray.shape)
# for i in range(0, len(X_train)):
#     X_train_gray[i] = rgb2gray(X_train[i])
# for i in range(0, len(X_valid)):
#     X_valid_gray[i] = rgb2gray(X_valid[i])
# for i in range(0, len(X_test)):
#     X_test_gray[i] = rgb2gray(X_test[i])

# # expanding dimensions for compatibility
# X_train_gray = np.expand_dims(X_train_gray, axis=3)
# X_valid_gray = np.expand_dims(X_valid_gray, axis=3)
# X_test_gray = np.expand_dims(X_test_gray, axis=3)

# print('Length of training : ', len(X_train) == len(X_train_gray))
# print('Length of validation : ', len(X_valid) == len(X_valid_gray))
# print('Length of testing : ', len(X_test) == len(X_test_gray))

# TODO: Remove this later
# sys.exit()

# normalzie the data
# convert image to YCbCr
from PIL import Image
def getYCbCr(rgb):
    img = Image.fromarray(rgb)
    img_yuv = img.convert('YCbCr')
    return img_yuv

# for i in range(0, len(X_train)):
#     X_train[i] = getYCbCr(X_train[i])
# for i in range(0, len(X_valid)):
#     X_valid[i] = getYCbCr(X_valid[i])
# for i in range(0, len(X_test)):
#     X_test[i] = getYCbCr(X_test[i])

# conver to grayscale and show
# save a grayscale image
img1 = Image.fromarray(X_train[2022].astype(np.uint8))
img1.save('normal.jpg')
img_ycbcr = getYCbCr(X_train[2022])
img_ycbcr = np.array(img_ycbcr)
img_ycbcr[:,:,0] *=0
img_ycbcr[:,:,1] *=0
img_ycbcr = Image.fromarray(img_ycbcr)
img_ycbcr.save('ycbcr.jpg')
img_gray = rgb2gray(X_train[2022])
img_gray = Image.fromarray(img_gray.astype(np.uint8))
img_gray.save('grayscale.jpg')
```

In [33]:
```
def normalize_img(img):
    return (img-128.0)/128.0


# for i in range(0, len(X_train)):
#     X_train[i] = (X_train[i]-128.0)/128.0

X_train = normalize_img(X_train)
X_valid = normalize_img(X_valid)
X_test = normalize_img(X_test)

# print('printing next one')
# print(X_train[0])

def shuffle_two(x, y):
    s = np.arange(x.shape[0])
    np.random.shuffle(s)
    return x[s], y[s]
```

Model Architecture

In [34]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.

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

EPOCHS = 10
BATCH_SIZE = 128
n_channels = 3 # grayscale or rgb used


def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for t
he weights and biases for each layer
    mu = 0
    sigma = 0.1
    dropout_prob = 0.75

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, n_channels, 16), me
an = mu, stddev = sigma))
    conv1_b = tf.Variable(tf.zeros(16))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID')
+ conv1_b

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

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

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
```

```
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 16, 32), mean = mu,
 stddev = sigma))
    conv2_b = tf.Variable(tf.zeros(32))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALI
D') + conv2_b

    # SOLUTION: Activation.
    conv2 = tf.nn.relu(conv2)

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

    # add additional convolution layer with strides of 2
    conv3_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 32, 64), mean = mu,
 stddev = sigma))
    conv3_b = tf.Variable(tf.zeros(64))
    conv3   = tf.nn.conv2d(conv2, conv3_W, strides=[1, 2, 2, 1], padding='VALI
D') + conv3_b

    conv3 = tf.nn.relu(conv3)

    # SOLUTION: Flatten. Input = 5x5x16. Output = 400.
    fc0   = flatten(conv3)

    # SOLUTION: Layer 3: Fully Connected. Input = 400. Output = 120.
    fc1_W = tf.Variable(tf.truncated_normal(shape=(576, 400), mean = mu, stdde
v = sigma))
    fc1_b = tf.Variable(tf.zeros(400))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    # add dropout layer
    drp1 = tf.nn.dropout(fc1, keep_prob=dropout_prob)

    # SOLUTION: Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W  = tf.Variable(tf.truncated_normal(shape=(400, 120), mean = mu, stdd
ev = sigma))
    fc2_b  = tf.Variable(tf.zeros(120))
    fc2    = tf.matmul(drp1, fc2_W) + fc2_b

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

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 10.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(120, n_classes), mean =
mu, stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros(n_classes))
    logits = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

In [35]:
```
# module for evaluation
x = tf.placeholder(tf.float32, (None, 32, 32, n_channels))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, n_classes)

# adding learning rate with decay
global_step = tf.Variable(0, trainable=False)
start_learning_rate = 0.001
decayed_learning_rate = tf.train.exponential_decay(start_learning_rate, global
_step,
                                                   1000000, 0.90, staircase=True)
rate = 0.001

logits = LeNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logi
ts=logits)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation, global_step=global_ste
p)

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, sess):
    num_examples = len(X_data)
    total_accuracy = 0
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:off
set+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batc
h_y})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

Train, Validate and Test the Model

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

In [36]:
```
### Train your model here.
### Calculate and report the accuracy on the training and validation set.
### Once a final model architecture is selected,
### the accuracy on the test set should be calculated and reported as well.
### Feel free to use as many code cells as needed.

config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session(config=config)

sess.run(tf.global_variables_initializer())
num_examples = len(X_train)

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

    validation_accuracy = evaluate(X_valid, y_valid, sess)
    print("EPOCH {} ...".format(i+1))
    print("Validation Accuracy = {:.3f}".format(validation_accuracy))

# check accuracy on test set
test_accuracy = evaluate(X_test, y_test, sess)
print("Test Accuracy at end = {:.3f}".format(test_accuracy))


# saving the model now
saver.save(sess, './model/traffic')
print("Model saved")
```

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.860
EPOCH 2 ...
Validation Accuracy = 0.902
EPOCH 3 ...
Validation Accuracy = 0.914
EPOCH 4 ...
Validation Accuracy = 0.930
EPOCH 5 ...
Validation Accuracy = 0.935
EPOCH 6 ...
Validation Accuracy = 0.936
EPOCH 7 ...
Validation Accuracy = 0.933
EPOCH 8 ...
Validation Accuracy = 0.935
EPOCH 9 ...
Validation Accuracy = 0.941
EPOCH 10 ...
Validation Accuracy = 0.941
Test Accuracy at end = 0.925
Model saved
```

Step 3: Test a Model on New Images

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

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

Load and Output the Images

In [40]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
# testing on the web_examples images
# load the web examples and create a dataset
from PIL import Image
import PIL

web_examples = './web_examples'
signs_dir = os.listdir(web_examples)
test_imgs = []
test_classes = []
for directory in signs_dir:
    img_paths = os.listdir(os.path.join(web_examples, directory))
    for img_path in img_paths:
        im = Image.open(os.path.join(web_examples, directory, img_path))
        res = im.resize((32,32), resample=PIL.Image.BILINEAR)
        im_arr = np.asarray(res)
        im_class = int(directory)
        test_classes.append(im_class)
        test_imgs.append(im_arr)

test_imgs = np.asarray(test_imgs)
test_imgs = normalize_img(test_imgs)
test_classes = np.asarray(test_classes)
print(test_imgs.shape)
print(test_classes.shape)
```

```
(15, 32, 32, 3)
(15,)
```

Predict the Sign Type for Each Image

In [41]:
```
### Run the predictions here and use the model to output the prediction for ea
ch image.
### Make sure to pre-process the images with the same pre-processing pipeline
 used earlier.
### Feel free to use as many code cells as needed.
# evaluate the model for testing


predictions = sess.run([tf.argmax(logits,1), y, correct_prediction, accuracy_o
peration], feed_dict={x: test_imgs, y: test_classes})
print(predictions[0])
print(predictions[1])
#print('Accuracy on web images = {:.3f} '.format(web_test_accuracy))
```

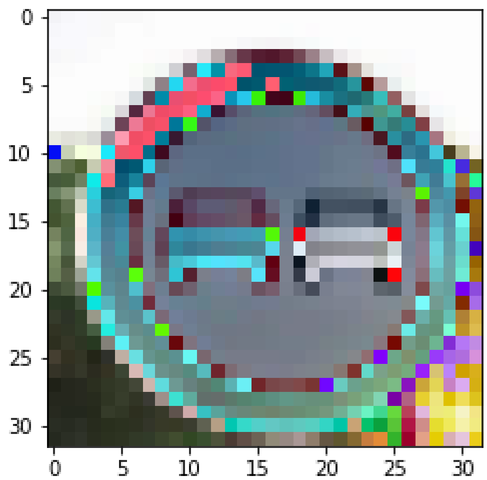
```
[13 13 13  0 10 22 41 22 18 18 38 38  1  9  9]
[13 13 13 22 22 22 22 22 27 27 38 38  6  9  9]
```

Analyze Performance

In [42]:
```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% a
ccurate on these new images.
print('Correct predictions are ', predictions[2])
print('Accuracy is ', predictions[3])
plt.imshow(test_imgs[13])
```

```
Correct predictions are  [ True  True  True False False  True False  True Fal
se False  True  True
 False  True  True]
Accuracy is  0.6
```

Out[42]: <matplotlib.image.AxesImage at 0x191519f39e8>



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

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

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

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

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

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.07893497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

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

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

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

In [43]:
```
### Print out the top five softmax probabilities for the predictions on the Ge
rman traffic sign images found on the web.
### Feel free to use as many code cells as needed.

top_probs = sess.run(tf.nn.top_k(logits, k=5), feed_dict={x: test_imgs, y: tes
t_classes})
for probs in top_probs:
    print(probs)
```

```
[[ 36.03691101  15.93601418   7.67464542   7.34629488   5.21829224]
 [ 59.88978577  32.13100433  15.36717033  15.26539135   8.49218178]
 [ 26.26918793  17.57087326  13.00547028   7.18609953   4.64785624]
 [ 17.73836136  16.00431252  13.17220974  10.76198387  10.72484589]
 [ 13.72799301   9.69709873   9.30805492   9.1125288    6.59511709]
 [ 27.65084839  13.34746552  11.78603649  11.49574757  11.43812943]
 [ 31.02389717  30.44897461  27.86355209  27.20183182  27.00528145]
 [ 48.64923096  24.3541069   11.05685997   8.81495476   6.621696  ]
 [ 37.06886292  18.9382267   13.04432774   8.76822567   7.36147738]
 [ 13.61449146  12.89269066  12.70088577   8.09046841   6.29207516]
 [ 58.97919846  11.15716362   9.53701878   9.36451626   6.19049215]
 [ 25.96920204  17.94626236  14.44877148  10.86267185   6.27249289]
 [  6.66532755   3.52831292   3.08595085   2.90443826   2.8545239 ]
 [ 41.62783813  21.5801239   18.93593788  17.08218956  12.44872189]
 [ 18.95693016  12.02656651  10.32982349   6.58629513   6.06086445]]
[[13  9  3 28 20]
 [13  9 10  3 15]
 [13  9 10  3  7]
 [ 0 32 11  1 28]
 [23 10 28 20  3]
 [22  4 15 20 17]
 [20 41 23  9 11]
 [22 26 20  4 15]
 [18  4  0 27 25]
 [18 31  4 25 20]
 [38 13 34 25  8]
 [38 34 13 17 30]
 [16  5 32 12  0]
 [ 9 19 35 13 12]
 [ 9 20 19 12 28]]
```

Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this [template (https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md)](https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project/blob/master/writeup_template.md) as a guide. The writeup can be in a markdown or pdf file.

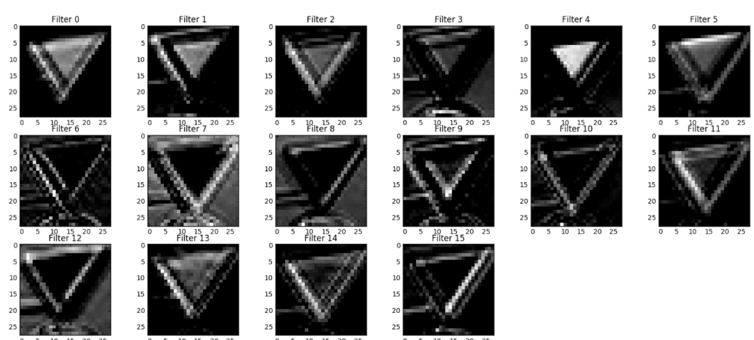
> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

Step 4 (Optional): Visualize the Neural Network's State with Test Images

This Section is not required to complete but acts as an additional excersise for understaning the output of a neural network's weights. While neural networks can be a great learning device they are often referred to as a black box. We can understand what the weights of a neural network look like better by plotting their feature maps. After successfully training your neural network you can see what it's feature maps look like by plotting the output of the network's weight layers in response to a test stimuli image. From these plotted feature maps, it's possible to see what characteristics of an image the network finds interesting. For a sign, maybe the inner network feature maps react with high activation to the sign's boundary outline or to the contrast in the sign's painted symbol.

Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the LeNet lab's (https://classroom.udacity.com/nanodegrees/nd013/parts/fbf77062-5703-404e-b60c-95b78b2f3f9e/modules/6df7ae49-c61c-4bb2-a23e-6527e69209ec/lessons/601ae704-1035-4287-8b11-e2c2716217ad/concepts/d4aca031-508f-4e0b-b493-e7b706120f81) feature maps looked like for it's second convolutional layer you could enter conv2 as the tf_activation variable.

For an example of what feature map outputs look like, check out NVIDIA's results in their paper End-to-End Deep Learning for Self-Driving Cars (https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/) in the section Visualization of internal CNN State. NVIDIA was able to show that their network's inner weights had high activations to road boundary lines by comparing feature maps from an image with a clear path to one without. Try experimenting with a similar test to show that your trained network's weights are looking for interesting features, whether it's looking at differences in feature maps from images with or without a sign, or even what feature maps look like in a trained network vs a completely untrained one on the same sign image.



Your output should look something like this (above)