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 [107]:
```
# Load pickled data
import pickle

# SOLUTION: Fill this in based on where you saved the training and testing dat
a

training_file = "traffic-signs-data/train.p"
validation_file= "traffic-signs-data/valid.p"
testing_file = "traffic-signs-data/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 [108]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
import numpy as np

# SOLUTION: Number of training examples
n_train = y_train.shape[0]

# SOLUTION: Number of validation examples
n_validation = y_valid.shape[0]

# SOLUTION: Number of testing examples.
n_test = y_test.shape[0]

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

# SOLUTION: How many unique classes/labels there are in the dataset.
n_classes = np.unique(y_train).shape[0]

print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("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 [109]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random
# Visualizations will be shown in the notebook.
%matplotlib inline
```

In [110]:
```
# Displaying 10 random images
number_of_images = 10

fig, axs = plt.subplots(2,5, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(number_of_images):
    index = random.randint(0, len(X_train))
    image = X_train[index]
    axs[i].axis('off')
    axs[i].imshow(image)
    axs[i].set_title(y_train[index])
```

In [ ]:

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.

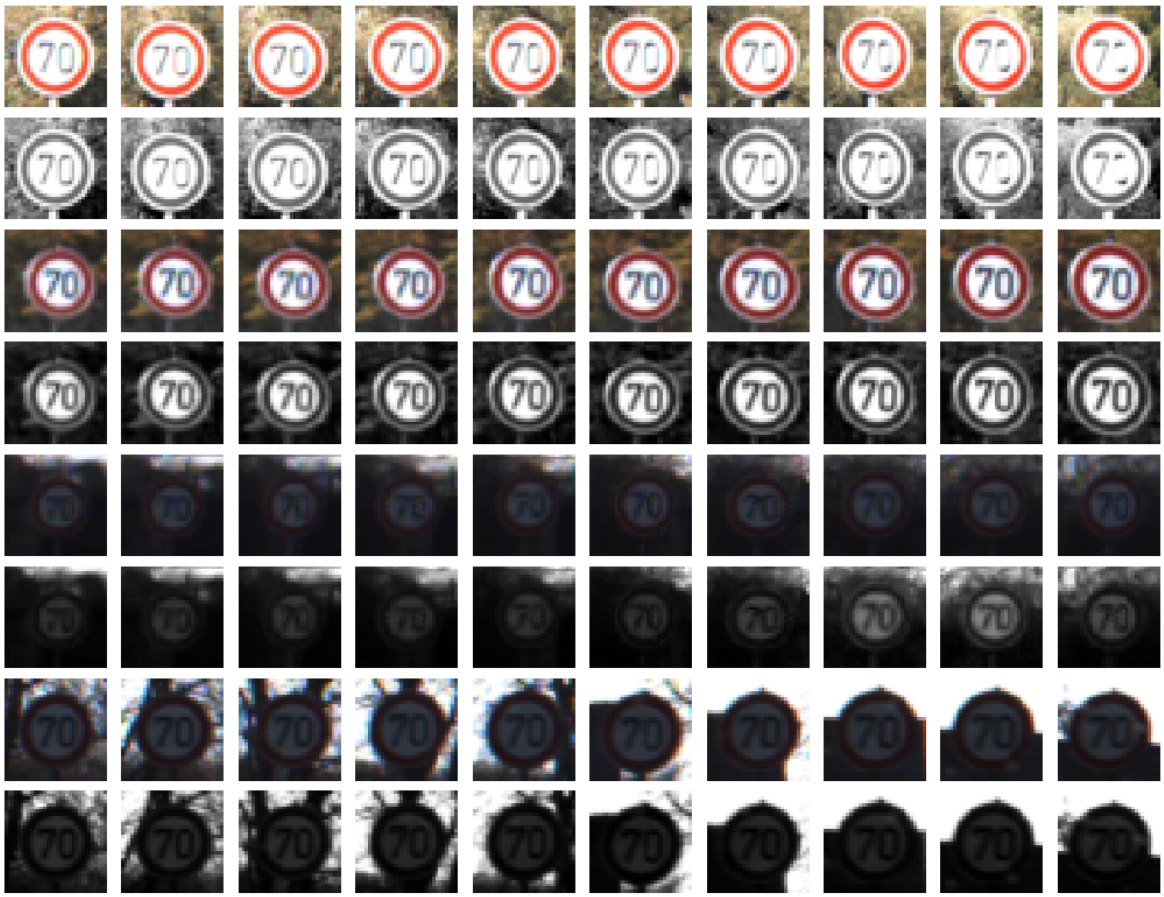
In [111]:
```
### Preprocess the data here. It is required to normalize the data. Other prep
rocessing steps could include
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.

# Converting images to grayscale
X_train_rgb = X_train
X_train_gray = np.sum(X_train/3, axis=3, keepdims=True)

X_test_rgb = X_test
X_test_gray = np.sum(X_test/3, axis=3, keepdims=True)

X_valid_gray = np.sum(X_valid/3,axis=3,keepdims=True)

print('RGB shape:', X_train_rgb.shape)
print('Grayscale shape:', X_train_gray.shape)
```

```
RGB shape: (34799, 32, 32, 3)
Grayscale shape: (34799, 32, 32, 1)
```

In [112]:
```
# Assigning newly processed images to training

X_train = X_train_gray/255
X_test = X_test_gray/255
X_valid = X_valid_gray/255

print('Grayscale images are assigned to training')
print('Grayscale shape:', X_valid.shape)
```

```
Grayscale images are assigned to training
Grayscale shape: (4410, 32, 32, 1)
```

In [113]:
```
# Visualize color and grayscale images

n_rows = 8
n_cols = 10
offset = 8000
fig, axs = plt.subplots(n_rows,n_cols, figsize=(18, 14))
fig.subplots_adjust(hspace = .1, wspace=.001)
axs = axs.ravel()
for j in range(0,n_rows,2):
    for i in range(n_cols):
        index = i + j*n_cols
        image = X_train_rgb[index + offset]
        axs[index].axis('off')
        axs[index].imshow(image)
    for i in range(n_cols):
        index = i + j*n_cols + n_cols
        image = X_train_gray[index + offset - n_cols].squeeze()
        axs[index].axis('off')
        axs[index].imshow(image, cmap='gray')
```



In [114]:
```
# Normalizing datasets to (-1,1)
#  This might be redunant steps. Should be removed.

X_train_normalized = (X_train - 128)/128
X_test_normalized = (X_test - 128)/128

print(np.mean(X_train_normalized))
print(np.mean(X_test_normalized))
```

```
-0.99746698563
-0.99748319668
```

In [115]:
```
#  This might be redunant steps. Should be removed.

print("Original shape:", X_train.shape)
print("Normalized shape:", X_train_normalized.shape)
fig, axs = plt.subplots(1,2, figsize=(10, 3))
axs = axs.ravel()

axs[0].axis('off')
axs[0].set_title('Original Image')
axs[0].imshow(X_train[0].squeeze(), cmap='gray')

axs[1].axis('off')
axs[1].set_title('Normalized Image')
axs[1].imshow(X_train_normalized[0].squeeze(), cmap='gray')
```

```
Original shape: (34799, 32, 32, 1)
Normalized shape: (34799, 32, 32, 1)
```

Out[115]: <matplotlib.image.AxesImage at 0x19c0234fb00>

In [116]:
```
from sklearn.preprocessing import LabelBinarizer

encoder = LabelBinarizer()
encoder.fit(y_train)
y_train = encoder.transform(y_train)
y_test = encoder.transform(y_test)
y_valid = encoder.transform(y_valid)

# Change to float32, so that it can be multiplied against the features in Tens
orFlow which are float32
y_train = y_train.astype(np.float32)
y_test = y_test.astype(np.float32)
y_valid = y_valid.astype(np.float32)
is_labels_encod = True

print('One hot encoding is done')
```

```
One hot encoding is done
```

Model Architecture

In [117]:
```
# My dataset preprocessing consisted of:
"1. Grayscale conversion 2. Normalizing to the range (-1, 1) - this is redunan
t 3. One hot encoding"
```

Out[117]:
```
'1. Grayscale conversion 2. Normalizing to the range (-1, 1) - this is reduna
nt 3. One hot encoding'
```

In [118]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.

# Copied from LeNet architecture from the module excercies
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

    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 6), mean = mu, s
tddev = sigma))
    conv1_b = tf.Variable(tf.zeros(6))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID')
+ conv1_b

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

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

    # Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu,
```

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

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

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

    # Flatten. Input = 5x5x16. Output = 400.
    fc0   = flatten(conv2)

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

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

    # Layer 4: Fully Connected. Input = 120. Output = 84.
    fc2_W = tf.Variable(tf.truncated_normal(shape=(120, 84), mean = mu, stddev
 = sigma))
    fc2_b = tf.Variable(tf.zeros(84))
    fc2   = tf.matmul(fc1, fc2_W) + fc2_b

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

    # Layer 5: Fully Connected. Input = 84. Output = n_classes (43).
    fc3_W = tf.Variable(tf.truncated_normal(shape=(84, n_classes), mean = mu,
stddev = sigma))
    fc3_b = tf.Variable(tf.zeros(n_classes))
    logits   = tf.matmul(fc2, fc3_W) + fc3_b

    return logits
```

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 [119]:
```
### 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.

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

from sklearn.utils import shuffle

testing = tf.constant('Testing Tensorflow. It is working')
sess = tf.Session()
print(sess.run(testing))
```

```
b'Testing Tensorflow. It is working'
```

In [120]:
```
# Most of the code is reused from the module excercies

x = tf.placeholder(tf.float32, (None, 32, 32, 1))
y = tf.placeholder(tf.int32, (None))
keep_prob = tf.placeholder(tf.float32)

# Defining global variables
EPOCHS = 50
BATCH_SIZE = 156
rate = 0.00097
mu = 0
sigma = 0.1

logits = LeNet(x)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=logit
s)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = rate)
training_operation = optimizer.minimize(loss_operation)

correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(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:off
set+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batc
h_y, keep_prob: 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [122]:
```
# Most of the code is reused from the module excercies

with tf.Session() as sess:

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

    validation_accuracy_figure = []
    train_accuracy_figure = []
    X_train, y_train = shuffle(X_train, y_train)

    for i in range(EPOCHS):
        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, ke
ep_prob: 0.5})

        validation_accuracy = evaluate(X_valid, y_valid)
        validation_accuracy_figure.append(validation_accuracy)

        train_accuracy = evaluate(X_train, y_train)
        train_accuracy_figure.append(train_accuracy)
        print("EPOCH {} ...".format(i+1))
        print("Train Accuracy = {:.3f}".format(train_accuracy))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

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

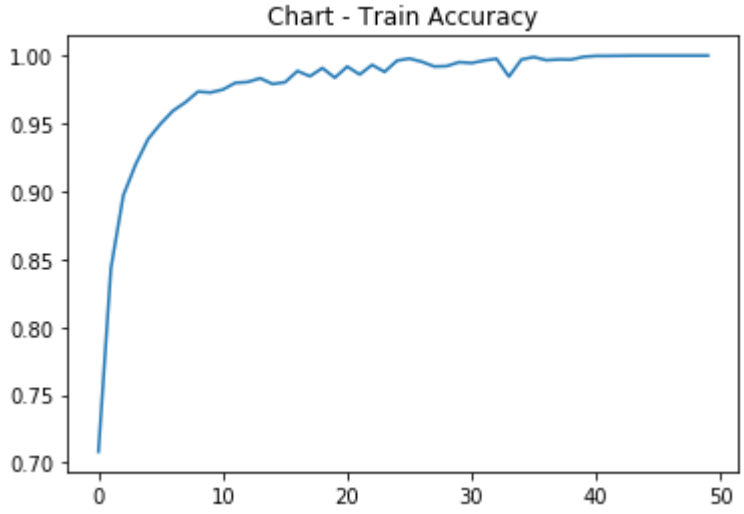
```
EPOCH 1 ...
Train Accuracy = 0.708
Validation Accuracy = 0.634

EPOCH 2 ...
Train Accuracy = 0.843
Validation Accuracy = 0.779

EPOCH 3 ...
Train Accuracy = 0.897
Validation Accuracy = 0.826

EPOCH 4 ...
Train Accuracy = 0.920
Validation Accuracy = 0.845

EPOCH 5 ...
Train Accuracy = 0.939
Validation Accuracy = 0.867

EPOCH 6 ...
Train Accuracy = 0.950
Validation Accuracy = 0.873

EPOCH 7 ...
Train Accuracy = 0.959
Validation Accuracy = 0.876

EPOCH 8 ...
Train Accuracy = 0.966
Validation Accuracy = 0.883

EPOCH 9 ...
Train Accuracy = 0.973
Validation Accuracy = 0.891

EPOCH 10 ...
Train Accuracy = 0.973
Validation Accuracy = 0.888

EPOCH 11 ...
Train Accuracy = 0.975
Validation Accuracy = 0.888

EPOCH 12 ...
Train Accuracy = 0.980
Validation Accuracy = 0.892

EPOCH 13 ...
Train Accuracy = 0.981
Validation Accuracy = 0.888

EPOCH 14 ...
Train Accuracy = 0.983
Validation Accuracy = 0.897

EPOCH 15 ...
```

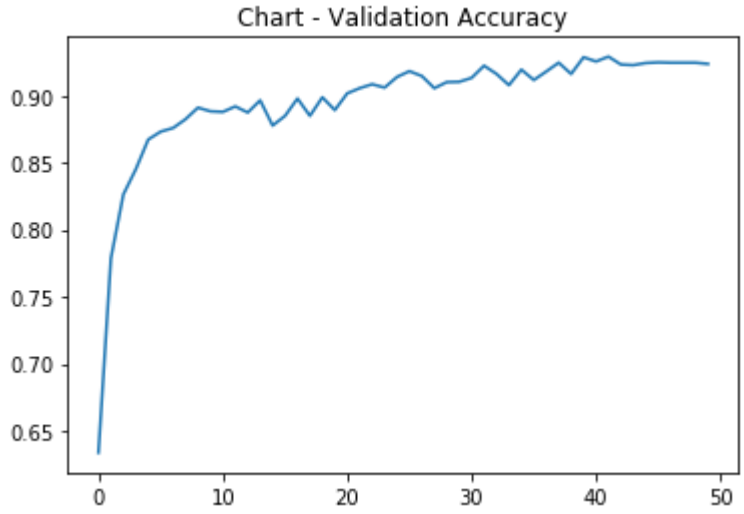
```
Train Accuracy = 0.979
Validation Accuracy = 0.878

EPOCH 16 ...
Train Accuracy = 0.980
Validation Accuracy = 0.885

EPOCH 17 ...
Train Accuracy = 0.989
Validation Accuracy = 0.898

EPOCH 18 ...
Train Accuracy = 0.985
Validation Accuracy = 0.885

EPOCH 19 ...
Train Accuracy = 0.991
Validation Accuracy = 0.899

EPOCH 20 ...
Train Accuracy = 0.984
Validation Accuracy = 0.889

EPOCH 21 ...
Train Accuracy = 0.992
Validation Accuracy = 0.902

EPOCH 22 ...
Train Accuracy = 0.986
Validation Accuracy = 0.906

EPOCH 23 ...
Train Accuracy = 0.993
Validation Accuracy = 0.909

EPOCH 24 ...
Train Accuracy = 0.988
Validation Accuracy = 0.906

EPOCH 25 ...
Train Accuracy = 0.996
Validation Accuracy = 0.914

EPOCH 26 ...
Train Accuracy = 0.998
Validation Accuracy = 0.918

EPOCH 27 ...
Train Accuracy = 0.995
Validation Accuracy = 0.915

EPOCH 28 ...
Train Accuracy = 0.992
Validation Accuracy = 0.906

EPOCH 29 ...
Train Accuracy = 0.992
```

```
Validation Accuracy = 0.910

EPOCH 30 ...
Train Accuracy = 0.995
Validation Accuracy = 0.910

EPOCH 31 ...
Train Accuracy = 0.995
Validation Accuracy = 0.913

EPOCH 32 ...
Train Accuracy = 0.996
Validation Accuracy = 0.922

EPOCH 33 ...
Train Accuracy = 0.998
Validation Accuracy = 0.916

EPOCH 34 ...
Train Accuracy = 0.985
Validation Accuracy = 0.908

EPOCH 35 ...
Train Accuracy = 0.997
Validation Accuracy = 0.920

EPOCH 36 ...
Train Accuracy = 0.999
Validation Accuracy = 0.912

EPOCH 37 ...
Train Accuracy = 0.997
Validation Accuracy = 0.918

EPOCH 38 ...
Train Accuracy = 0.997
Validation Accuracy = 0.925

EPOCH 39 ...
Train Accuracy = 0.997
Validation Accuracy = 0.916

EPOCH 40 ...
Train Accuracy = 0.999
Validation Accuracy = 0.929

EPOCH 41 ...
Train Accuracy = 1.000
Validation Accuracy = 0.926

EPOCH 42 ...
Train Accuracy = 1.000
Validation Accuracy = 0.929

EPOCH 43 ...
Train Accuracy = 1.000
Validation Accuracy = 0.923
```

```
EPOCH 44 ...
Train Accuracy = 1.000
Validation Accuracy = 0.923

EPOCH 45 ...
Train Accuracy = 1.000
Validation Accuracy = 0.924

EPOCH 46 ...
Train Accuracy = 1.000
Validation Accuracy = 0.925

EPOCH 47 ...
Train Accuracy = 1.000
Validation Accuracy = 0.925

EPOCH 48 ...
Train Accuracy = 1.000
Validation Accuracy = 0.925

EPOCH 49 ...
Train Accuracy = 1.000
Validation Accuracy = 0.925

EPOCH 50 ...
Train Accuracy = 1.000
Validation Accuracy = 0.924

Model saved
```

In [123]:
```
plt.plot(train_accuracy_figure)
plt.title("Chart - Train Accuracy")
plt.show()

plt.plot(validation_accuracy_figure)
plt.title("Chart - Validation Accuracy")
plt.show()

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    train_accuracy = evaluate(X_train, y_train)
    print("Train Accuracy = {:.3f}".format(train_accuracy))
    valid_accuracy = evaluate(X_valid, y_valid)
    print("Valid Accuracy = {:.3f}".format(valid_accuracy))
```






```
Train Accuracy = 1.000
Valid Accuracy = 0.924
```

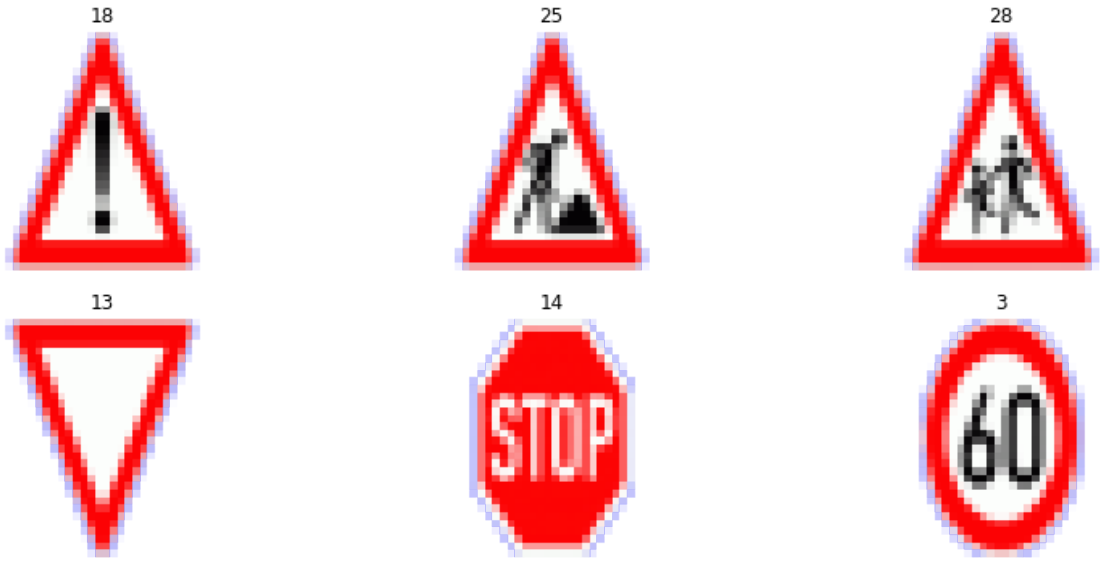
Step 3: Test a Model on New Images

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

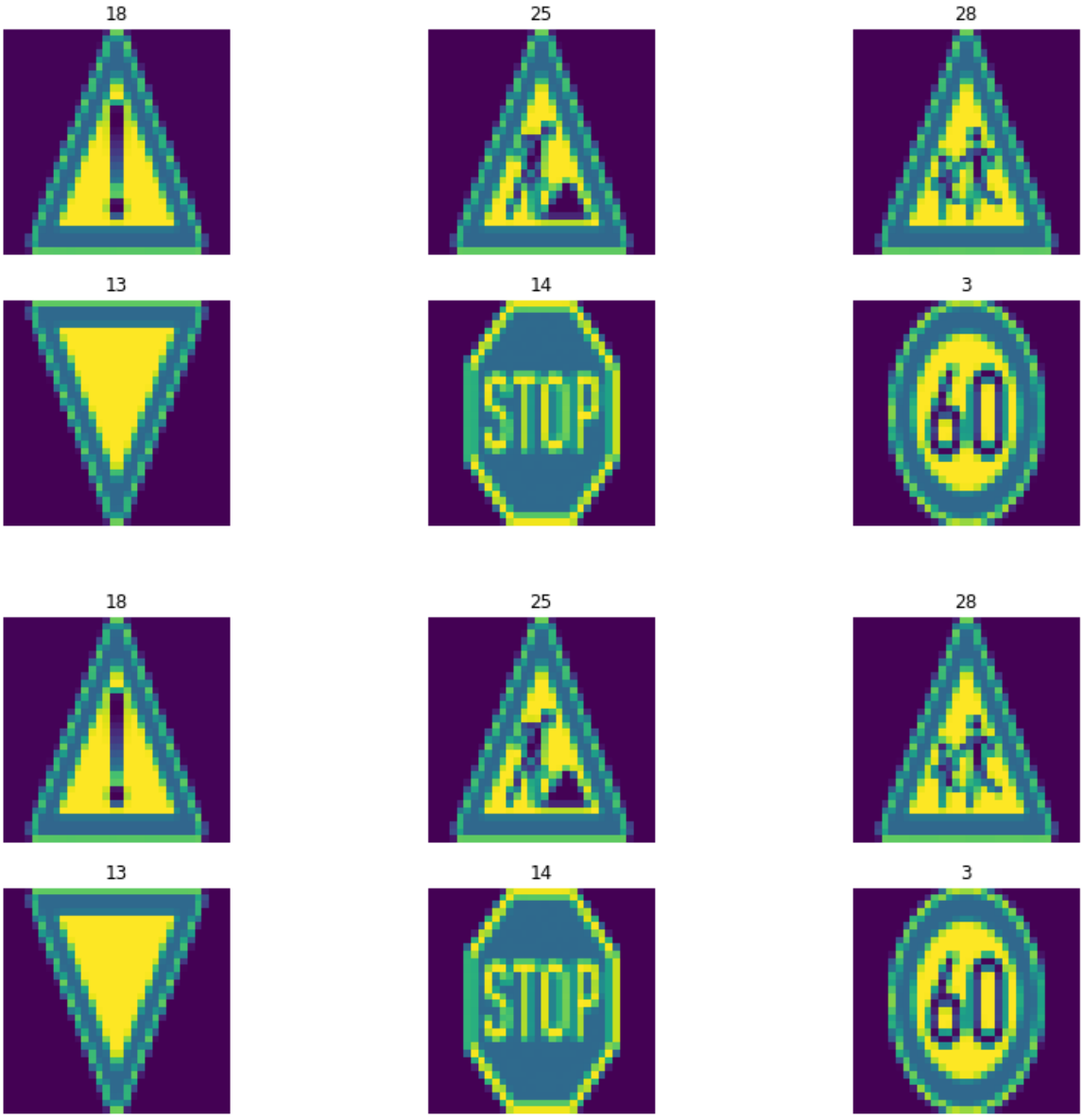
In [124]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.

import glob
import cv2
import matplotlib.image as mpimg

my_images = sorted(glob.glob('./german-signs-data/*.gif'))
my_labels = np.array([18, 25, 28, 13, 14, 3])

my_signs = []
my_signs_gray = []
my_signs_normalized = []

# Displaying all acquired german signs
# This could be bundled into a function because this has been used above
number_of_images = 6

fig, axs = plt.subplots(2, 3, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(number_of_images):
    index = my_labels[i]
    image = mpimg.imread(my_images[i])
    image = cv2.resize(image, (32,32), interpolation=cv2.INTER_AREA)
    my_signs.append(image)
    axs[i].axis('off')
    axs[i].imshow(image)
    axs[i].set_title(index)
```



Predict the Sign Type for Each Image

In [126]:
```
### 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.

# Converting images to grayscale
my_signs = np.array(my_signs)
my_signs_gray = np.sum(my_signs/3, axis=3, keepdims=True)
my_signs_normalized = my_signs_gray/255

print('RGB shape:', my_signs.shape)
print('Grayscale shape:', my_signs_gray.shape)
print('Normalized shape:', my_signs_normalized.shape)

# Displaying all acquired german signs
# This could be bundled into a function because this has been used above
number_of_images = 6

fig, axs = plt.subplots(2, 3, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(number_of_images):
    index = my_labels[i]
    image = my_signs_gray[i]
    image = cv2.resize(image, (32,32), interpolation=cv2.INTER_AREA)
    axs[i].axis('off')
    axs[i].imshow(image)
    axs[i].set_title(index)


fig, axs = plt.subplots(2, 3, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(number_of_images):
    index = my_labels[i]
    image = my_signs_normalized[i]
    image = cv2.resize(image, (32,32), interpolation=cv2.INTER_AREA)
    axs[i].axis('off')
    axs[i].imshow(image)
    axs[i].set_title(index)
```

```
RGB shape: (6, 32, 32, 4)
Grayscale shape: (6, 32, 32, 1)
Normalized shape: (6, 32, 32, 1)
```



In [127]:
```
my_labels = encoder.transform(my_labels)
my_labels = my_labels.astype(np.float32)


print('Labels shape:', my_labels.shape)
print('Normalized shape:', my_signs_normalized.shape)
```

```
Labels shape: (6, 43)
Normalized shape: (6, 32, 32, 1)
```

In [128]:
```
correct_predictions={}
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess, "./traffic_signs")
    num_examples = len(my_signs)
    total_accuracy = 0

    for ind in range(0, num_examples):
        correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(y, 1))
        correct_predictions[ind] = correct_prediction
        accuracy = sess.run(accuracy_operation, feed_dict={x: my_signs_normali
zed, y: my_labels, keep_prob: 1.0})
```

Analyze Performance

In [129]:
```
### 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("Data Set Accuracy = {:.3f}".format(accuracy))
```

```
Data Set Accuracy = 0.833
```

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 [131]:
```
### 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.

softmax_logits = tf.nn.softmax(logits)
top_k = tf.nn.top_k(softmax_logits, k = 5)
name_values = np.genfromtxt('signnames.csv', skip_header=1, dtype=[('myint','i
8'), ('mysring','S55')], delimiter=',')

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.restore(sess,"./traffic_signs")
    my_softmax_logits = sess.run(softmax_logits, feed_dict={x: my_signs_normal
ized, keep_prob: 1.0})
    my_top_k = sess.run(top_k, feed_dict={x: my_signs_normalized, keep_prob:
1.0})

    for i in range(len(my_signs)):
        guess_probs = my_top_k[0][i]
        guess_labels = my_top_k[1][i]
        guess_names = name_values[guess_labels]
        print('predicted signs:')
        print(guess_names)
        print('with corresponding probabilities: ')
        print(guess_probs)
        print("======================")
```

```
predicted signs:
[(18, b'General caution') (26, b'Traffic signals') (12, b'Priority road')
 (27, b'Pedestrians') (40, b'Roundabout mandatory')]
with corresponding probabilities:
[  1.00000000e+00   6.11154349e-10   1.02727260e-18   4.94604789e-19
   9.91444077e-20]

predicted signs:
[(25, b'Road work') (20, b'Dangerous curve to the right')
 (38, b'Keep right') (36, b'Go straight or right') (23, b'Slippery road')]
with corresponding probabilities:
[  1.00000000e+00   1.10255146e-20   6.15212405e-26   9.59336039e-28
   7.91629206e-32]

predicted signs:
[(28, b'Children crossing') (20, b'Dangerous curve to the right')
 (41, b'End of no passing') (32, b'End of all speed and passing limits')
 ( 0, b'Speed limit (20km/h)')]
with corresponding probabilities:
[  9.85252142e-01   1.47478916e-02   3.76155712e-16   5.44196897e-17
   3.43958606e-17]

predicted signs:
[(13, b'Yield') (35, b'Ahead only') (36, b'Go straight or right')
 (12, b'Priority road') ( 0, b'Speed limit (20km/h)')]
with corresponding probabilities:
[  1.00000000e+00   5.98428144e-20   3.30372981e-32   1.56983333e-38
   0.00000000e+00]

predicted signs:
[(14, b'Stop') ( 4, b'Speed limit (70km/h)') ( 2, b'Speed limit (50km/h)')
 ( 1, b'Speed limit (30km/h)') (38, b'Keep right')]
with corresponding probabilities:
[  1.00000000e+00   2.57240301e-13   2.16597397e-13   4.07860933e-26
   3.21921686e-26]

predicted signs:
[( 2, b'Speed limit (50km/h)') ( 1, b'Speed limit (30km/h)')
 (31, b'Wild animals crossing') ( 3, b'Speed limit (60km/h)')
 ( 5, b'Speed limit (80km/h)')]
with corresponding probabilities:
[  9.99995351e-01   4.68261851e-06   1.99782804e-13   3.66856954e-19
   2.06883229e-24]
```

In [132]:
```
"Last image was classifed wrongly. Instead of 60 km/h, it was classified as 50
 km/h"
```

Out[132]:
```
'Last image was classifed wrongly. Instead of 60 km/h, it was classified as 5
0 km/h'
```

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) as a guide. The writeup can be in a markdown or pdf file.

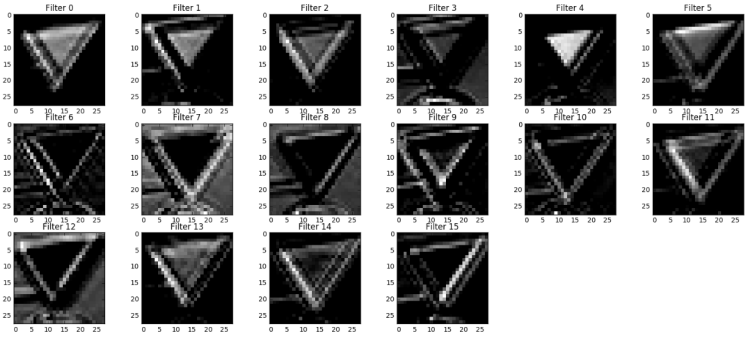
> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**.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

In [ ]:
```
### Visualize your network's feature maps here.
### Feel free to use as many code cells as needed.

# image_input: the test image being fed into the network to produce the featur
e maps
# tf_activation: should be a tf variable name used during your training proced
ure that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more deta
il, by default matplot sets min and max to the actual min and max values of th
e output
# plt_num: used to plot out multiple different weight feature map sets on the
 same block, just extend the plt number for each new feature map entry

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation
_max=-1 ,plt_num=1):
    # Here make sure to preprocess your image_input in a way your network expe
cts
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data placeh
older variable
    # If you get an error tf_activation is not defined it may be having troubl
e accessing the variable from inside a function
    activation = tf_activation.eval(session=sess,feed_dict={x : image_input})
    featuremaps = activation.shape[3]
    plt.figure(plt_num, figsize=(15,15))
    for featuremap in range(featuremaps):
        plt.subplot(6,8, featuremap+1) # sets the number of feature maps to sh
ow on each row and column
        plt.title('FeatureMap ' + str(featuremap)) # displays the feature map
 number
        if activation_min != -1 & activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
 vmin =activation_min, vmax=activation_max, cmap="gray")
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
 vmax=activation_max, cmap="gray")
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
 vmin=activation_min, cmap="gray")
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest",
 cmap="gray")
```