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 [1]:

```python
# Load pickled data
import pickle
import cv2
import tensorflow as tf

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

training_file = 'train.p'
validation_file= 'valid.p'
testing_file = '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']

print('Data imported')
```

```
Data imported
```

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 [2]:

```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
import numpy as np
from collections import Counter
# TODO: Number of training examples
n_train = len(X_train)

# 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(np.unique(y_train))
labels, count_signs_by_class = zip(*Counter(y_train).items())

mean = np.mean(X_train)
std = np.std(X_train)

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

```
Number of training examples = 34799
Number of testing examples = 12630
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.

In [3]:

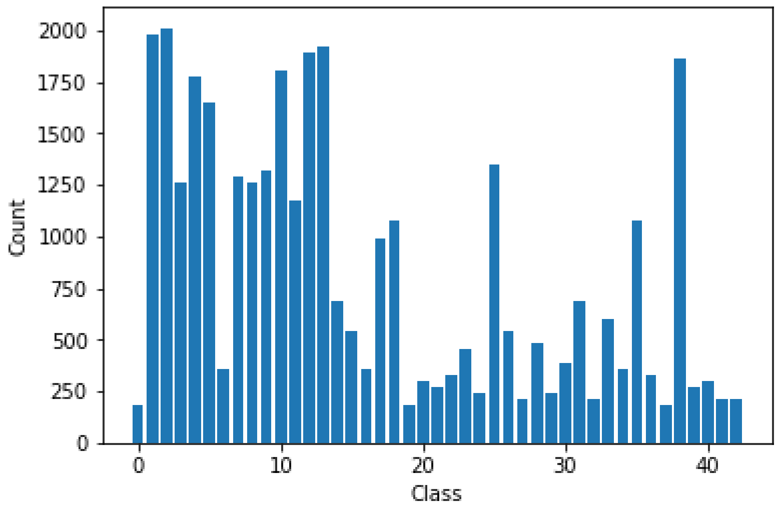
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random
# Visualizations will be shown in the notebook.
%matplotlib inline

# Show 25 random images from
fig_random, axs_random = plt.subplots(5,5, figsize=(10, 10))
fig_random.subplots_adjust(hspace = .35)
axs = axs_random.ravel()
for i in range(25):
    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 [4]:

```
# Plot count of classes

signs = np.arange(n_classes)
labels_train = y_train.tolist()
sign_types = [labels_train.count(c) for c in signs]

fig_class, axs_class = plt.subplots()
axs_class.set_ylabel('Count')
axs_class.set_xlabel('Class')
plt.bar(signs, sign_types)
plt.show
```

Out[4]:

```
<function matplotlib.pyplot.show>
```



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.)

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

Model Architecture

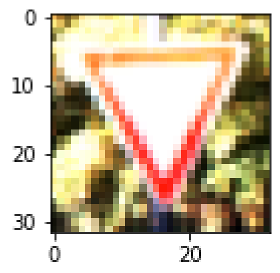
```
In [5]:
```

```python

from sklearn.model_selection import train_test_split
X_train_non_normalized, y_train_non_normalized = train['features'], train['labels']
X_test, y_test = test['features'], test['labels']

print("Training: ", len(X_train_non_normalized))
print("Test: ", len(X_test))

X_train_non_normalized, X_validation, y_train_non_normalized, y_validation = train_test_spl

print("Updated Image Shape: {}".format(X_train_non_normalized[0].shape))
```

```
Training:  34799
Test:  12630
Updated Image Shape: (32, 32, 3)
```

In [6]:

```
### Preprocess the data here. Preprocessing steps could include normalization, converting t
### Feel free to use as many code cells as needed.
import cv2
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from skimage.color import rgb2gray
import numpy as np

X_train = []
y_train = []

def augment_brightness(image):
    image = cv2.cvtColor(image,cv2.COLOR_RGB2HSV)
    image = np.array(image, dtype = np.float64)
    random_bright = .5+np.random.uniform()
    image[:,:,2] = image[:,:,2]*random_bright
    image[:,:,2][image[:,:,2]>255]  = 255
    image = np.array(image, dtype = np.uint8)
    image = cv2.cvtColor(image,cv2.COLOR_HSV2RGB)
    return image

def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

def normalize(img):
    return cv2.normalize(img, img, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV

for i, (image, label) in enumerate(zip(X_train_non_normalized, y_train_non_normalized)):
    brightness_image = augment_brightness(image)
    norm_image = normalize(brightness_image)
    #gray_scale = grayscale(norm_image)
    X_train.append(norm_image)
    y_train.append(label)

index = random.randint(0, len(X_train))

image = X_train[index]
plt.figure(figsize=(2,2))
plt.imshow(image, cmap='gray')

# Show a random collection of the preprocessed images
fig = plt.figure()
fig.set_figwidth(16)
fig.set_figheight(10)

for index, sign in enumerate(signs):
    ax = fig.add_subplot(5, 10, index + 1)
    image = X_train[random.randint(0, len(X_train) - 1)].squeeze()
    ax.imshow(image, cmap='gray')
```



In [7]:

```
from sklearn.utils import shuffle

X_train, y_train = shuffle(X_train, y_train)
```

In [8]:

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

def LeNet(x):
    # Hyperparameters
    mu = 0
    sigma = 0.1

    # SOLUTION: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 3, 6), mean = mu, stddev = sigma
    conv1_b = tf.Variable(tf.zeros(6))
    conv1   = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

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

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

    # SOLUTION: Layer 2: Convolutional. Output = 10x10x16.
    conv2_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 6, 16), mean = mu, stddev = sigm
    conv2_b = tf.Variable(tf.zeros(16))
    conv2   = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

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

    # Addtional dropout
    conv2 = tf.nn.dropout(conv2, keep_prob)

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

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

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

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

    # SOLUTION: 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

    # SOLUTION: Activation and dropout
    fc2    = tf.nn.relu(fc2)
    fc2    = tf.nn.dropout(fc2, keep_prob)

    # SOLUTION: Layer 5: Fully Connected. Input = 84. Output = 43.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros(43))
    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 [9]:

```
x = tf.placeholder(tf.float32,(None, 32, 32, 3))
y = tf.placeholder(tf.int32, (None))
keep_prob = tf.placeholder(tf.float32)
one_hot_y = tf.one_hot(y,43)


### Train your model here.
EPOCHS = 50
BATCH_SIZE = 128
rate = 0.005

logits = LeNet(x)
labels_predicted = tf.nn.softmax(logits)

cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits = logits, labels = one_hot_y
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(one_hot_y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()

def evaluate(X_data, y_data):
    num_examples = len(X_data)
    total_accuracy = 0
    sess = tf.get_default_session()
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offset+BATCH_SIZ
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y, keep_pro
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples

print("Finished")
```

```
Finished
```

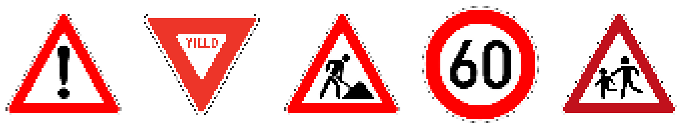
In [10]:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_examples = len(X_train)

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train, y_train = shuffle(X_train, y_train)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train[offset:end], y_train[offset:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep_prob: 0.8}

        validation_accuracy = evaluate(X_validation, y_validation)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        training_accuracy = evaluate(X_train, y_train)
        print("Training Accuracy = {:.3f}".format(training_accuracy))
        print()

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

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

```
Validation Accuracy = 0.959
Training Accuracy = 0.991

EPOCH 47 ...
Validation Accuracy = 0.881
Training Accuracy = 0.969

EPOCH 48 ...
Validation Accuracy = 0.971
Training Accuracy = 0.993

EPOCH 49 ...
Validation Accuracy = 0.948
Training Accuracy = 0.990

EPOCH 50 ...
Validation Accuracy = 0.966
Training Accuracy = 0.990

Test Accuracy = 0.904
```

In [13]:

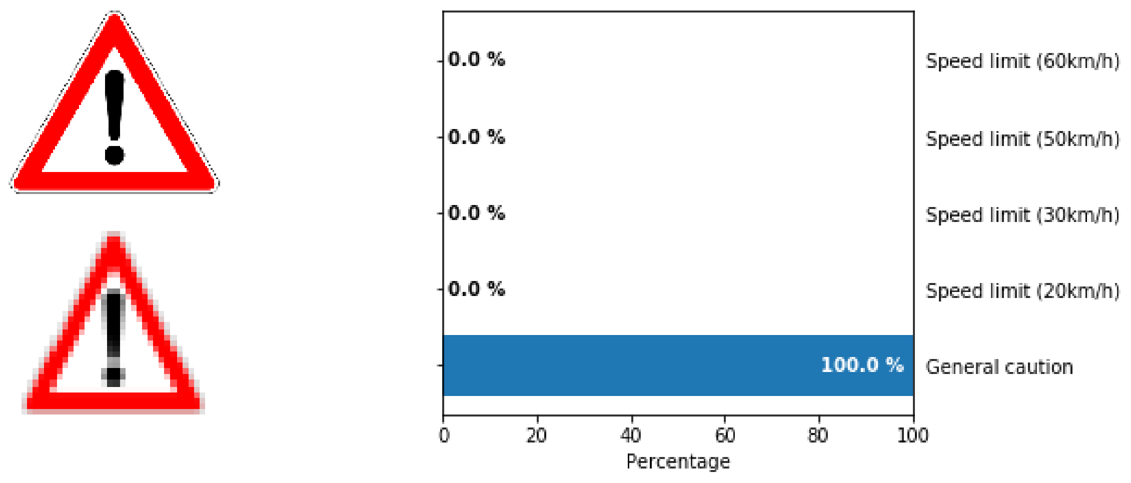
```
# Signs downloaded from internet
import glob
image_files = glob.glob('testimages/*.jpg')
image_count = len(image_files)
test_data = np.zeros((image_count, 32, 32, 3), dtype=np.uint8)
fig, ax = plt.subplots(1, image_count)
for i in range(image_count):
    img = plt.imread(image_files[i])
    ax[i].imshow(img, cmap='gray')
    ax[i].axis('off')
    img = cv2.resize(img,  (32, 32), interpolation=cv2.INTER_AREA)
    test_data[i] = img
```



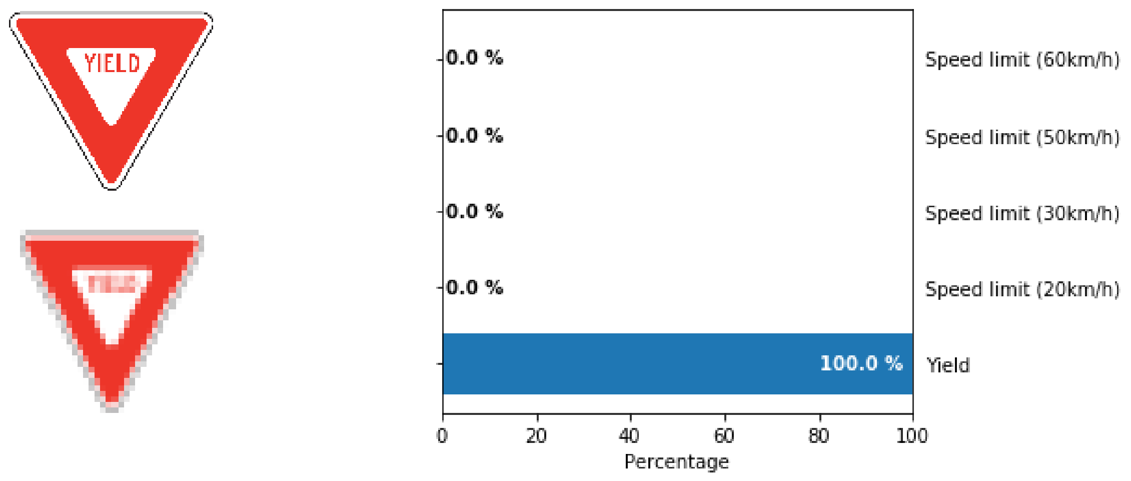
Predict the Sign Type for Each Image

In [14]:

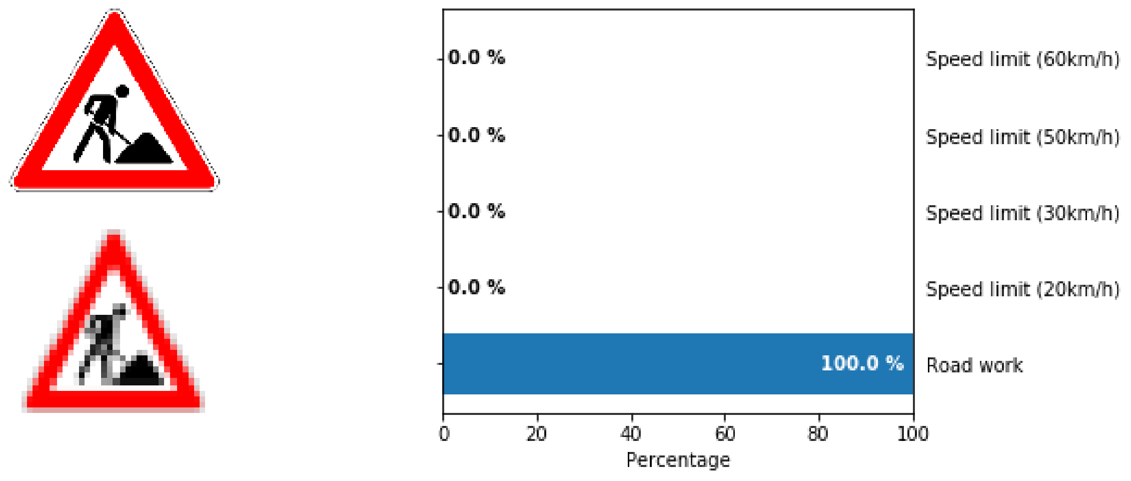
```
### Run the predictions here and use the model to output the prediction for each image.
### Make sure to pre-process the images with the same pre-processing pipeline used earlier.
### Feel free to use as many code cells as needed.
import matplotlib.gridspec as gridspec
import pandas as pd

signnames = pd.read_csv("signnames.csv").values[:, 1]

with tf.Session() as sess:
    saver = tf.train.Saver()
    saver.restore(sess, save_path='./models/lenet')

    # Calculate the top 5 predictions for each example image
    top_5 = tf.nn.top_k(labels_predicted, 5)
    top_5_predictions = sess.run(top_5, feed_dict={x: test_data, keep_prob: 1.0})

    # Show each example image and the corresponding top 5 predictions
    for i in range(len(image_files)):
        # Define grid
        plt.figure(figsize = (10, 4))
        gs = gridspec.GridSpec(2, 2)

        # Show original image
        plt.subplot2grid((2, 2), (0, 0), colspan=1, rowspan=1)
        plt.imshow(plt.imread(image_files[i]))
        plt.axis('off')

        # Show preprocessed image
        plt.subplot2grid((2, 2), (1, 0), colspan=1, rowspan=1)
        plt.imshow(test_data[i].squeeze(), cmap='gray')
        plt.axis('off')

        # Show predictions
        ax = plt.subplot2grid((2, 2), (0, 1), colspan=2, rowspan=2)
        ax.set_xlim([0, 100])

        rects = ax.barh(np.arange(5) + 0.5, [value * 100.0 for value in top_5_predictions[0
        ax.set_yticks(np.arange(5) + 0.5)
        ax.set_yticklabels(signnames[top_5_predictions[1][i].astype(int)])
        ax.set_xlabel('Percentage')
        plt.tick_params(axis='both', which='both', labelleft='off', labelright='on', labelt

        for index, rect in enumerate(rects):
            width = int(rect.get_width())
            xValue = round(top_5_predictions[0][i][index] * 100.0, 2)
            labelStr = str(xValue) + ' %'

            if (width < 25):
                xloc = width + 1
                clr = 'black'
                align = 'left'
            else:
                xloc = 0.98 * width
                clr = 'white'
                align = 'right'

            yloc = rect.get_y() + rect.get_height() / 2.0
            ax.text(xloc, yloc, labelStr, horizontalalignment=align, verticalalignment='cer
                            color=clr, weight='bold', clip_on=True)
    plt.show()
```



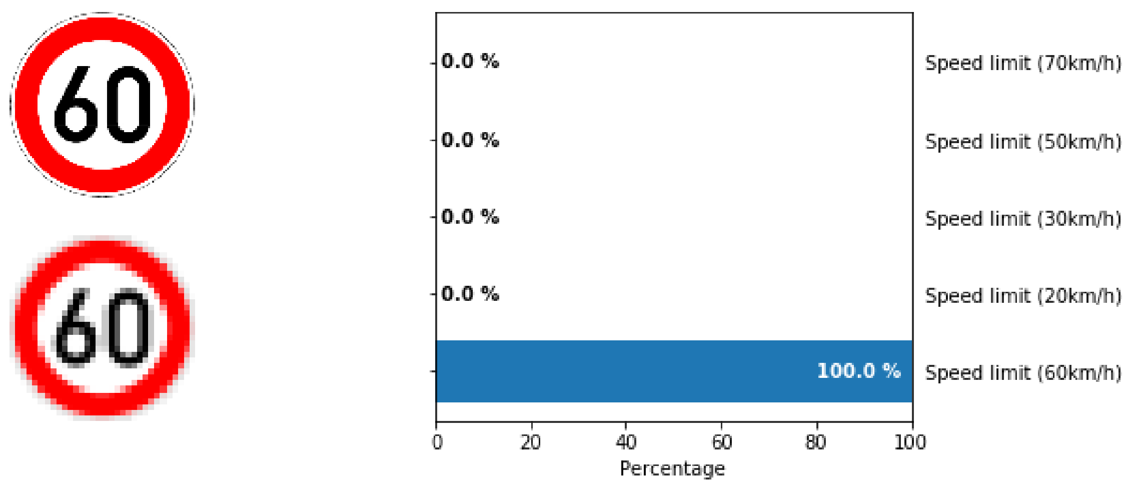
0.0 %
0.0 %
0.0 %
0.0 %
100.0 %
Speed limit (60km/h)
Speed limit (50km/h)
Speed limit (30km/h)
Speed limit (20km/h)
General caution
0
20
40
60
80
100
Percentage

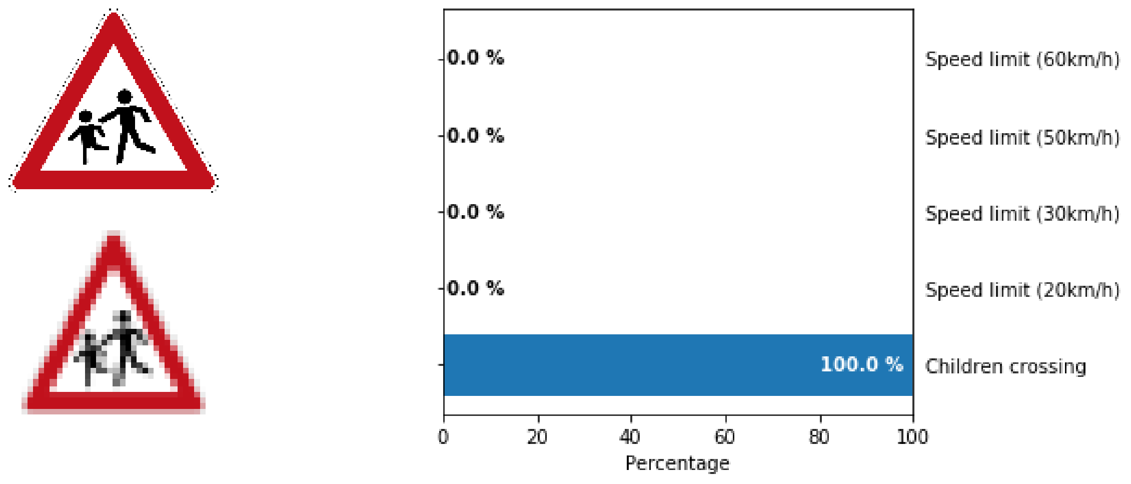


YIELD
0.0 %
0.0 %
0.0 %
0.0 %
100.0 %
Speed limit (60km/h)
Speed limit (50km/h)
Speed limit (30km/h)
Speed limit (20km/h)
Yield
0
20
40
60
80
100
Percentage



0.0 %
0.0 %
0.0 %
0.0 %
100.0 %
Speed limit (60km/h)
Speed limit (50km/h)
Speed limit (30km/h)
Speed limit (20km/h)
Road work
0
20
40
60
80
100
Percentage





Analyze Performance

In [ ]:

```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate on th
```

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. `tk.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 [ ]:

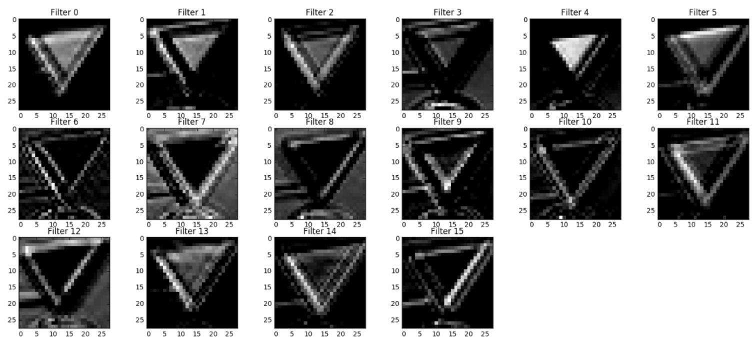
```
### Print out the top five softmax probabilities for the predictions on the German traffic
### Feel free to use as many code cells as needed.
```

Step 4: 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 feature maps
# tf_activation: should be a tf variable name used during your training procedure that repr
# activation_min/max: can be used to view the activation contrast in more detail, by defaul
# plt_num: used to plot out multiple different weight feature map sets on the same block, j

def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_max=-1 ,plt_
    # Here make sure to preprocess your image_input in a way your network expects
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data placeholder variabl
    # If you get an error tf_activation is not defined it maybe having trouble accessing th
    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 show on each ro
        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 =activa
        elif activation_max != -1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax=activat
        elif activation_min !=-1:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin=activat
        else:
            plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap="gray")
```

Question 9

Discuss how you used the visual output of your trained network's feature maps to show that it had learned to look for interesting characteristics in traffic sign images

Answer:

> **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.

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.