Traffic_Sign_Classifier

August 3, 2017

1 Self-Driving Car Engineer Nanodegree

1.1 Deep Learning

1.2 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.

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 that can be used to guide the writing process. Completing the code template and writeup template will cover all of the rubric points for this project.

The rubric 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.

1.3 Step 0: Load The Data

```
In [1]: # Load pickled data
        import pickle
        import cv2


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

        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']

print("Load data complete")
```

```
Load data complete
```

1.4 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 might be useful for calculating some of the summary results.

1.4.1 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

        # 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 = set(y_train)

        print("Number of training examples =", n_train)
        print("Number of valication examples =", n_validation)
        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 valication examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
```

1.4.2 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.

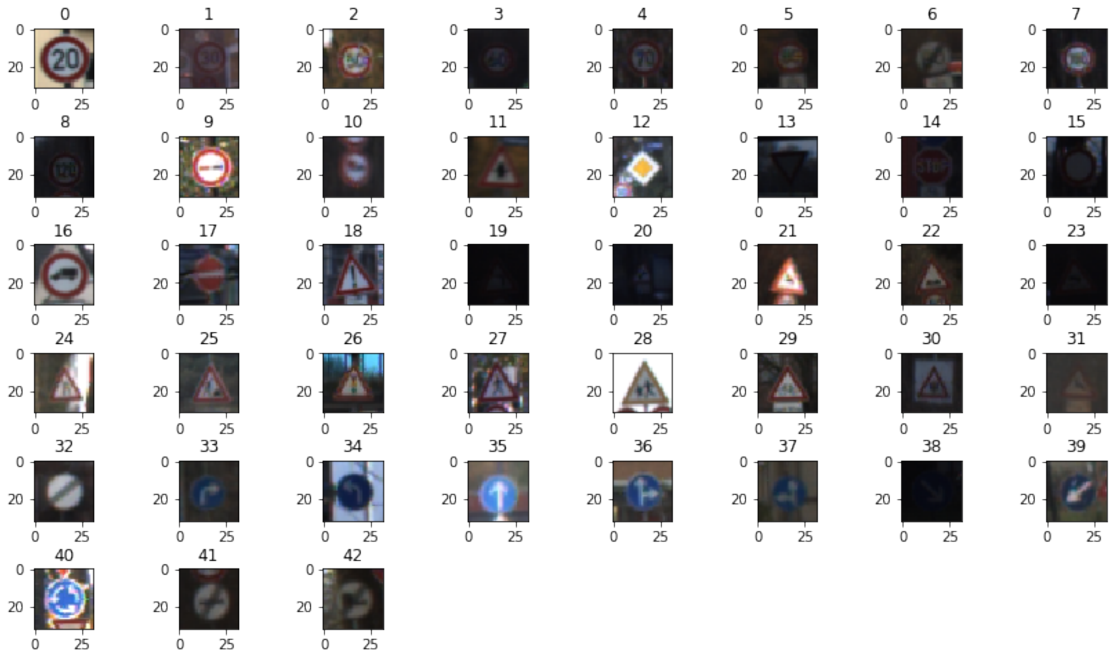
The Matplotlib examples and gallery pages are a great resource for doing visualizations in Python.

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?

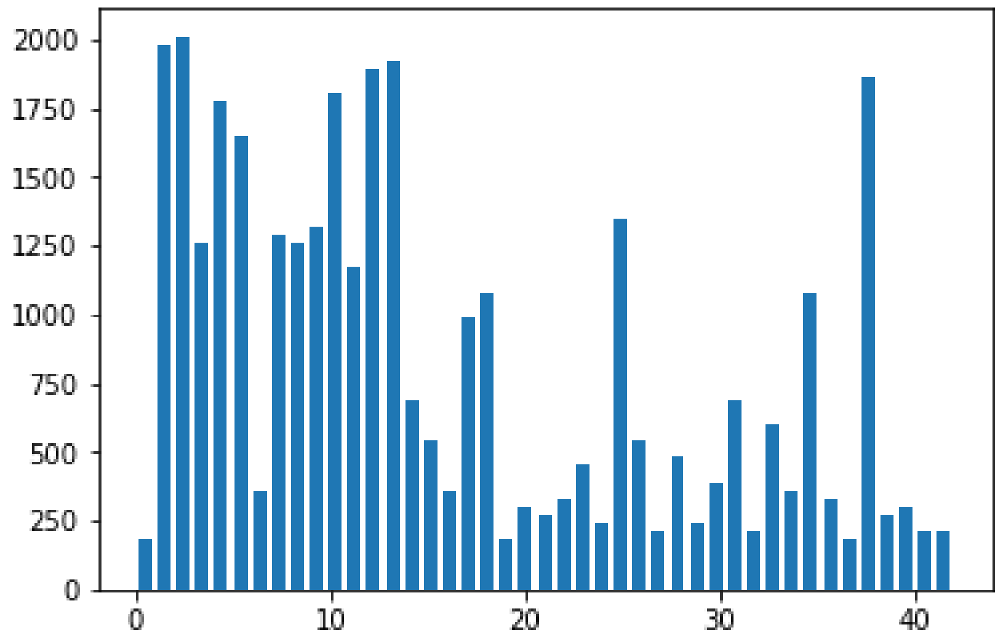
```
In [3]: ### Data exploration visualization code goes here.
        ### Feel free to use as many code cells as needed.
        import matplotlib.pyplot as plt
        import tensorflow as tf
        from random import *
        # Visualizations will be shown in the notebook.
        %matplotlib inline
```

```
In [4]: # Show a random sign in training data
        import numpy as np
        import matplotlib.image as mpimg


        fig, axs = plt.subplots(6,8, figsize=(15, 8))
        fig.subplots_adjust(hspace = .8, wspace=.01)
        axs = axs.ravel()
```

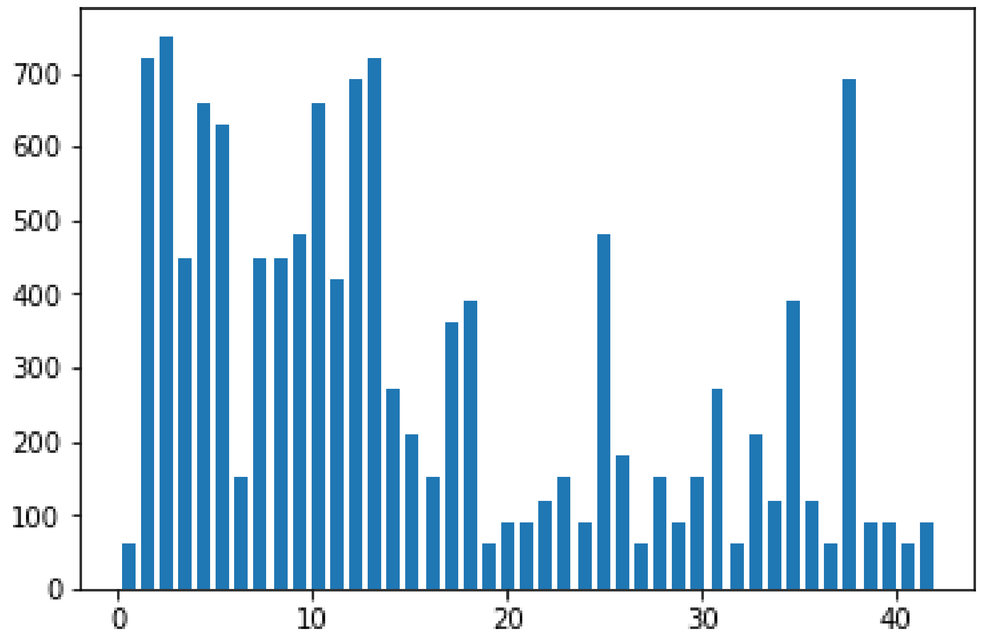
```
for i in list(n_classes):
    for j, j_ele in enumerate(y_train):
        if int(i) == int(j_ele):
            axs[i].imshow(X_train[j])
            axs[i].set_title(i)
            break

# Delete blank images
for i in range(43,48):
    fig.delaxes(axs[i])
```



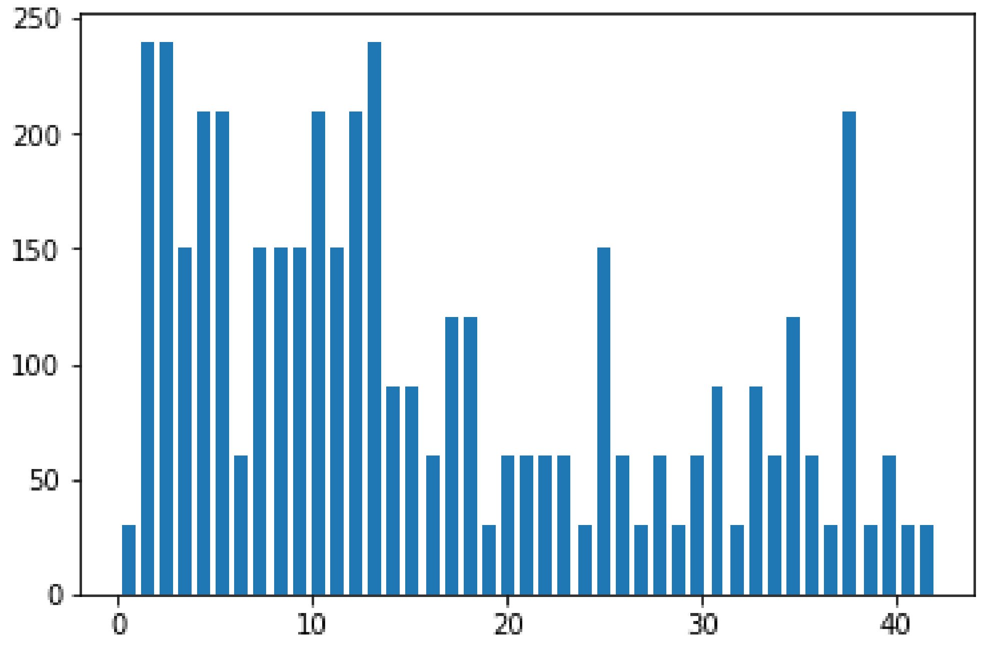
```
In [5]: # Make sure that the distributions of classes in train, test and validation data are the
        # histogram of label frequency on training data
        hist, bins = np.histogram(y_train, bins=len(n_classes))
        width = 0.7 * (bins[1] - bins[0])
        center = (bins[:-1] + bins[1:]) / 2
        plt.bar(center, hist, align='center', width=width)
        plt.show()
```



```
In [6]: # histogram of label frequency on test data
        hist, bins = np.histogram(y_test, bins=len(n_classes))
        width = 0.7 * (bins[1] - bins[0])
        center = (bins[:-1] + bins[1:]) / 2
        plt.bar(center, hist, align='center', width=width)
        plt.show()
```



```
In [7]: # histogram of label frequency on validation data
        hist, bins = np.histogram(y_valid, bins=len(n_classes))
        width = 0.7 * (bins[1] - bins[0])
        center = (bins[:-1] + bins[1:]) / 2
        plt.bar(center, hist, align='center', width=width)
        plt.show()
```



1.5 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.

The LeNet-5 implementation shown in the classroom 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. 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.

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

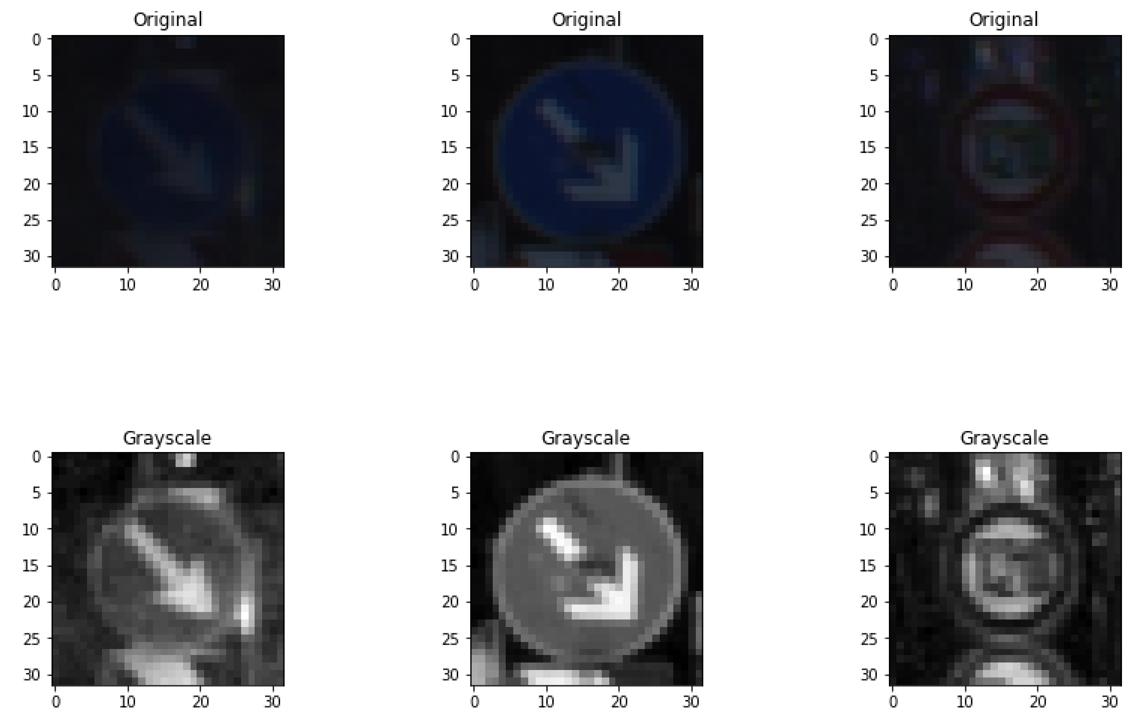
Minimally, the image data should be normalized so that the data has mean zero and equal variance. For image data, `(pixel - 128)/ 128` is a quick way to approximately normalize the data and can be used in this project.

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 [8]: # Grayscale
        img_size = X_train.shape[1]

        # Convert to grayscale
        X_train_rgb = X_train
        X_train_gry = np.sum(X_train/3, axis=3, keepdims=True)

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

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

        print(X_train_gry.shape)

        fig, axs = plt.subplots(2,3, figsize=(15, 8))
        fig.subplots_adjust(hspace = .8, wspace=.01)
        axs = axs.ravel()
        image_num = []
        for i in range(3):
            random_image_no = np.random.randint(X_train.shape[0], size=1)[0]
            image_num.append(random_image_no)
            axs[i].imshow(X_train[random_image_no])
            axs[i].set_title('Original')


        for i in range(3):
            axs[i+3].imshow(X_train_gry[image_num[i]].reshape(32,32), cmap = plt.get_cmap('gray'
            axs[i+3].set_title('Grayscale')

        print('RGB shape:', X_train_rgb.shape)
        print('Grayscale shape:', X_train_gry.shape)

        X_train = X_train_gry
        X_test = X_test_gry
        X_valid = X_valid_gry
```

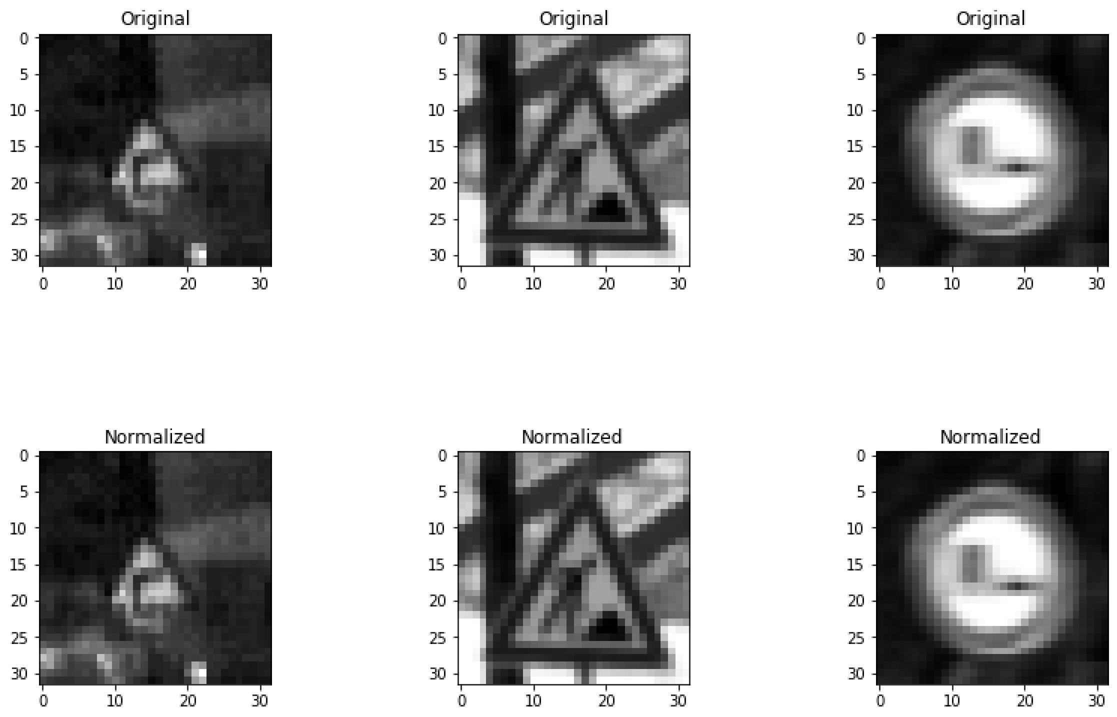
```
(34799, 32, 32, 1)
RGB shape: (34799, 32, 32, 3)
Grayscale shape: (34799, 32, 32, 1)
```



```
In [29]: # Normalize the train and test datasets to (-1,1)
         X_train_normalized = (X_train - 128)/128
         X_test_normalized = (X_test - 128)/128
         X_valid_normalized = (X_valid - 128)/128

         fig, axs = plt.subplots(2,3, figsize=(15, 8))
         fig.subplots_adjust(hspace = .8, wspace=.01)
         axs = axs.ravel()
         image_num = []
         for i in range(3):
             random_image_no = np.random.randint(X_train.shape[0], size=1)[0]
             image_num.append(random_image_no)
             axs[i].imshow(X_train[random_image_no].reshape(32,32), cmap = plt.get_cmap('gray'))
             axs[i].set_title('Original')
```

```
for i in range(3):
    axs[i+3].imshow(X_train_normalized[image_num[i]].reshape(32,32), cmap = plt.get_cma
    axs[i+3].set_title('Normalized')


print(np.mean(X_train_normalized))
print(np.mean(X_test_normalized))
X_train = X_train_normalized
X_test = X_test_normalized
X_valid = X_valid_normalized
```

```
-1.00276626043
-1.00279855589
```



1.5.2 Model Architecture

```
In [10]: # Store layers weight & bias
         mu = 0
         sigma = 0.1
         weights = {
             'wc1': tf.Variable(tf.random_normal([5, 5, 1, 6], mean = mu, stddev = sigma)),
             'wc2': tf.Variable(tf.random_normal([5, 5, 6, 16], mean = mu, stddev = sigma)),
             'wd1': tf.Variable(tf.random_normal([5*5*16, 400], mean = mu, stddev = sigma)),
             'wd2': tf.Variable(tf.random_normal([400, 256], mean = mu, stddev = sigma)),
             'wd3': tf.Variable(tf.random_normal([256, 120], mean = mu, stddev = sigma)),
             'wd4': tf.Variable(tf.random_normal([120, 84], mean = mu, stddev = sigma)),
             'wd5': tf.Variable(tf.random_normal([84, 43], mean=mu, stddev=sigma))}

         biases = {
             'bc1': tf.Variable(tf.zeros([6])),
             'bc2': tf.Variable(tf.zeros([16])),
             'bd1': tf.Variable(tf.zeros([0])),
             'bd2': tf.Variable(tf.zeros([256])),
             'bd3': tf.Variable(tf.zeros([120])),
             'bd4': tf.Variable(tf.zeros([84])),
             'bd5': tf.Variable(tf.zeros([43]))}
```

```
In [11]: ### Define your architecture here.
         ### Feel free to use as many code cells as needed.
         def Traffic_sign_model(x):
             # Arguments used for tf.truncated_normal, randomly defines variables for the weight

             # TODO: Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.
             conv = tf.nn.conv2d(x, weights['wc1'], strides=[1, 1, 1, 1], padding='VALID', name=
             conv = tf.nn.bias_add(conv, biases['bc1'])
             # TODO: Activation.
             conv = tf.nn.relu(conv, name='conv1_relu')
             # TODO: Pooling. Input = 28x28x6. Output = 14x14x6.
             maxpooling = tf.nn.max_pool(conv, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding
             # TODO: Layer 2: Convolutional. Output = 10x10x16.
             conv2 = tf.nn.conv2d(maxpooling, weights['wc2'], strides=[1, 1, 1, 1], padding='VAL
             conv2 = tf.nn.bias_add(conv2, biases['bc2'])
             # TODO: Activation.
             conv2 = tf.nn.relu(conv2,name='conv2_relu')
             # TODO: Pooling. Input = 10x10x16. Output = 5x5x16.
             maxpooling = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], paddin
             # TODO: Flatten. Input = 5x5x16. Output = 400.
             fc1 = tf.reshape(maxpooling, [-1, weights['wd1'].get_shape().as_list()[0]])
             # TODO: Layer 3: Fully Connected. Input = 400. Output = 256.
             fc1 = tf.add(tf.matmul(fc1, weights['wd2']), biases['bd2'])
             # TODO: Activation.
             fc1 = tf.nn.relu(fc1)
             # TODO: Layer 4: Fully Connected. Input = 256. Output = 120.
             fc2 = tf.add(tf.matmul(fc1, weights['wd3']), biases['bd3'])
             # TODO: Activation.
             fc2 = tf.nn.relu(fc2)

             # TODO: Layer 5: Fully Connected. Input = 120. Output = 84.
             fc3 = tf.add(tf.matmul(fc2, weights['wd4']), biases['bd4'])
```

```
    # TODO: Layer 6: Fully Connected. Input = 84. Output = 43.
    logits = tf.add(tf.matmul(fc3, weights['wd5']), biases['bd5'])


    return logits

from tensorflow.contrib.layers import flatten
keep_prob = tf.placeholder(tf.float32)
```

```
In [12]: # tf.reset_default_graph()
         x = tf.placeholder(tf.float32, (None, 32, 32, 1))
         y = tf.placeholder(tf.int32, (None))
         one_hot_y = tf.one_hot(y, depth=43, on_value=1., off_value=0., axis=-1)
```

1.5.3 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 [13]: ### 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.
         rate = 0.001

         logits = Traffic_sign_model(x)#Traffic_sign_model(x)
         cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=one_hot_y, logits=logits
         loss_operation = tf.reduce_mean(cross_entropy)
         optimizer = tf.train.AdamOptimizer(learning_rate = rate)
         # optimizer = tf.train.RMSPropOptimizer(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()
```

```
In [14]: # Model Evaluation
         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
                 accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: batch_y, keep
                 total_accuracy += (accuracy * len(batch_x))
             return total_accuracy / num_examples
```

```
In [15]: from sklearn.utils import shuffle

         EPOCHS = 100
         BATCH_SIZE = 128
         with tf.Session() as sess:
             sess.run(tf.global_variables_initializer())
             num_examples = X_train.shape[0]
             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:

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

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

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.829

EPOCH 2 ...
Validation Accuracy = 0.886

EPOCH 3 ...
Validation Accuracy = 0.906

EPOCH 4 ...
Validation Accuracy = 0.910

EPOCH 5 ...
Validation Accuracy = 0.924

EPOCH 6 ...
Validation Accuracy = 0.922

EPOCH 7 ...
Validation Accuracy = 0.936

EPOCH 8 ...
```

```
Validation Accuracy = 0.937

EPOCH 9 ...
Validation Accuracy = 0.927

EPOCH 10 ...
Validation Accuracy = 0.916

EPOCH 11 ...
Validation Accuracy = 0.932

EPOCH 12 ...
Validation Accuracy = 0.918

EPOCH 13 ...
Validation Accuracy = 0.934

EPOCH 14 ...
Validation Accuracy = 0.911

EPOCH 15 ...
Validation Accuracy = 0.941

EPOCH 16 ...
Validation Accuracy = 0.928

EPOCH 17 ...
Validation Accuracy = 0.940

EPOCH 18 ...
Validation Accuracy = 0.952

EPOCH 19 ...
Validation Accuracy = 0.928

EPOCH 20 ...
Validation Accuracy = 0.934

EPOCH 21 ...
Validation Accuracy = 0.945

EPOCH 22 ...
Validation Accuracy = 0.946

EPOCH 23 ...
Validation Accuracy = 0.929

EPOCH 24 ...
```

```
Validation Accuracy = 0.930

EPOCH 25 ...
Validation Accuracy = 0.940

EPOCH 26 ...
Validation Accuracy = 0.955

EPOCH 27 ...
Validation Accuracy = 0.957

EPOCH 28 ...
Validation Accuracy = 0.945

EPOCH 29 ...
Validation Accuracy = 0.936

EPOCH 30 ...
Validation Accuracy = 0.952

EPOCH 31 ...
Validation Accuracy = 0.941

EPOCH 32 ...
Validation Accuracy = 0.948

EPOCH 33 ...
Validation Accuracy = 0.959

EPOCH 34 ...
Validation Accuracy = 0.954

EPOCH 35 ...
Validation Accuracy = 0.959

EPOCH 36 ...
Validation Accuracy = 0.959

EPOCH 37 ...
Validation Accuracy = 0.960

EPOCH 38 ...
Validation Accuracy = 0.960

EPOCH 39 ...
Validation Accuracy = 0.959

EPOCH 40 ...
```

```
Validation Accuracy = 0.960

EPOCH 41 ...
Validation Accuracy = 0.959

EPOCH 42 ...
Validation Accuracy = 0.959

EPOCH 43 ...
Validation Accuracy = 0.959

EPOCH 44 ...
Validation Accuracy = 0.960

EPOCH 45 ...
Validation Accuracy = 0.961

EPOCH 46 ...
Validation Accuracy = 0.960

EPOCH 47 ...
Validation Accuracy = 0.961

EPOCH 48 ...
Validation Accuracy = 0.961

EPOCH 49 ...
Validation Accuracy = 0.963

EPOCH 50 ...
Validation Accuracy = 0.962

EPOCH 51 ...
Validation Accuracy = 0.961

EPOCH 52 ...
Validation Accuracy = 0.962

EPOCH 53 ...
Validation Accuracy = 0.962

EPOCH 54 ...
Validation Accuracy = 0.961

EPOCH 55 ...
Validation Accuracy = 0.909

EPOCH 56 ...
```

```
Validation Accuracy = 0.948

EPOCH 57 ...
Validation Accuracy = 0.955

EPOCH 58 ...
Validation Accuracy = 0.956

EPOCH 59 ...
Validation Accuracy = 0.951

EPOCH 60 ...
Validation Accuracy = 0.933

EPOCH 61 ...
Validation Accuracy = 0.931

EPOCH 62 ...
Validation Accuracy = 0.953

EPOCH 63 ...
Validation Accuracy = 0.944

EPOCH 64 ...
Validation Accuracy = 0.956

EPOCH 65 ...
Validation Accuracy = 0.930

EPOCH 66 ...
Validation Accuracy = 0.942

EPOCH 67 ...
Validation Accuracy = 0.946

EPOCH 68 ...
Validation Accuracy = 0.940

EPOCH 69 ...
Validation Accuracy = 0.947

EPOCH 70 ...
Validation Accuracy = 0.946

EPOCH 71 ...
Validation Accuracy = 0.934

EPOCH 72 ...
```

```
Validation Accuracy = 0.948

EPOCH 73 ...
Validation Accuracy = 0.944

EPOCH 74 ...
Validation Accuracy = 0.946

EPOCH 75 ...
Validation Accuracy = 0.949

EPOCH 76 ...
Validation Accuracy = 0.953

EPOCH 77 ...
Validation Accuracy = 0.956

EPOCH 78 ...
Validation Accuracy = 0.946

EPOCH 79 ...
Validation Accuracy = 0.952

EPOCH 80 ...
Validation Accuracy = 0.946

EPOCH 81 ...
Validation Accuracy = 0.943

EPOCH 82 ...
Validation Accuracy = 0.956

EPOCH 83 ...
Validation Accuracy = 0.948

EPOCH 84 ...
Validation Accuracy = 0.950

EPOCH 85 ...
Validation Accuracy = 0.945

EPOCH 86 ...
Validation Accuracy = 0.958

EPOCH 87 ...
Validation Accuracy = 0.938

EPOCH 88 ...
```

```
Validation Accuracy = 0.942

EPOCH 89 ...
Validation Accuracy = 0.961

EPOCH 90 ...
Validation Accuracy = 0.951

EPOCH 91 ...
Validation Accuracy = 0.955

EPOCH 92 ...
Validation Accuracy = 0.958

EPOCH 93 ...
Validation Accuracy = 0.960

EPOCH 94 ...
Validation Accuracy = 0.960

EPOCH 95 ...
Validation Accuracy = 0.960

EPOCH 96 ...
Validation Accuracy = 0.960

EPOCH 97 ...
Validation Accuracy = 0.959

EPOCH 98 ...
Validation Accuracy = 0.959

EPOCH 99 ...
Validation Accuracy = 0.959

EPOCH 100 ...
Validation Accuracy = 0.959

Model saved
```

```
In [16]: # Keras ImageDataGenerator to augment the images, but failed to improve the performance

         # from sklearn.utils import shuffle
         # from keras.preprocessing.image import ImageDataGenerator

         # datagen = ImageDataGenerator(
         #     featurewise_center=True,
         #     featurewise_std_normalization=True,
         #     rotation_range=20,
         #     width_shift_range=0.2,
         #     height_shift_range=0.2,
         #     horizontal_flip=True)
         # EPOCHS = 60
         # BATCH_SIZE = 32
         # with tf.Session() as sess:
         #     sess.run(tf.global_variables_initializer())
         #     num_examples = X_train.shape[0]
         #     print("Training...")
         #     print()
         #     datagen.fit(X_train)
         #     for i in range(EPOCHS):
         # #         X_train, y_train = shuffle(X_train, y_train)
         #         j = 0
         #         for batch_x, batch_y in datagen.flow(X_train, y_train, batch_size=BATCH_SIZE)
         # #             print(batch_x.shape)
         #             if j %1000 == 0:
         #                 print(j)
         # #             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
         #             j += 1
         #             if j > 3000:
         #                 break
         # #         validation_accuracy = evaluate(X_train, y_train)
         # #         print("EPOCH {} ...".format(i+1))
         # #         print("train Accuracy = {:.3f}".format(validation_accuracy))
         # #         print()
         #         validation_accuracy = evaluate(X_valid, y_valid)
         #         print("EPOCH {} ...".format(i+1))
         #         print("Validation Accuracy = {:.3f}".format(validation_accuracy))
         #         print()

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

```
In [17]: # saver = tf.train.Saver()
         EPOCHS = 100
         BATCH_SIZE = 128

         with tf.Session() as sess:
         #     sess.run(tf.global_variables_initializer())
             saver = tf.train.import_meta_graph('./lenet.meta')
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             test_accuracy = evaluate(X_test, y_test)
             print("Test Accuracy = {:.3f}".format(test_accuracy))
```

```
Test Accuracy = 0.938
```

2 Log

1. July 24 2017. Grayscale & Normalize
2. July 25 2017. Build a simple model. The train and validate acc is 0.058.
3. July 27 2017. Build the LeNet model. The train and validate acc is 0.050. Learning rate = 0.1 is too large.
4. July 28 2017. Rewrite the Grayscale and normalize part. The train and validate acc is almost 0.900. It's not good enough. Increasing the batch size and epochs failed to improve the performance. RMSPropOptimizer fails to improve the performance.
5. July 28 2017. Dropout = 1.0 -> 0.5 failed to improve the performance.
6. July 29 2017. ImageDataGenerator in keras to augment the images. The acc is about 0.55.
7. July 30 2017. Replace the ImageDataGenerator with simple preprocessing technique. Add a Dense layer to the LeNet network. Train more epochs = 100. The test acc is 0.932-0.936.
8. July 31 2017. Try to augment the images.

2.1 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.

2.1.1 Load and Output the Images

```
In [18]: ### Load the images and plot them here.
         ### Feel free to use as many code cells as needed.
         import os
         from skimage import transform,data
         import matplotlib.pyplot as plt


         def load_images_from_folder(folder):
             """
             load all the images in test folder.
             """
             images = []
             for filename in os.listdir(folder):
                 if filename.startswith('.'):
                     continue
                 img = plt.imread(os.path.join(folder,filename))
                 if img is not None:
                     dst = transform.resize(img,(32,32,3))
                     images.append(dst)
```

```
        return np.array(images)


    images = load_images_from_folder('traffic-signs-data/web_images')
    num_images = len(images)


    X_tmp = np.sum(images/3, axis=3, keepdims=True)
    X_images = (np.array(X_tmp) - 128)/128
    y_images = np.array([11,26,12,33,26])
```

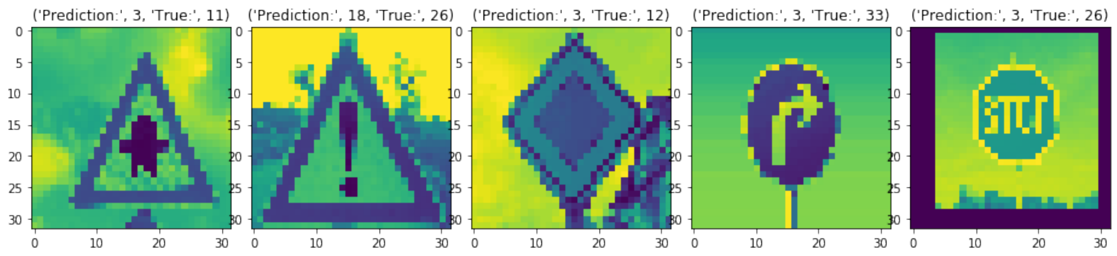
```
/Users/yy/anaconda/lib/python3.5/site-packages/skimage/transform/_warps.py:84: UserWarning: The
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/Users/yy/anaconda/lib/python3.5/site-packages/skimage/transform/_warps.py:84: UserWarning: The
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/Users/yy/anaconda/lib/python3.5/site-packages/skimage/transform/_warps.py:84: UserWarning: The
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/Users/yy/anaconda/lib/python3.5/site-packages/skimage/transform/_warps.py:84: UserWarning: The
  warn("The default mode, 'constant', will be changed to 'reflect' in "
/Users/yy/anaconda/lib/python3.5/site-packages/skimage/transform/_warps.py:84: UserWarning: The
  warn("The default mode, 'constant', will be changed to 'reflect' in "
```

2.1.2 Predict the Sign Type for Each Image

```
In [19]: ### 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 earl
         ### Feel free to use as many code cells as needed.


         # Load the model
         # Later, launch the model, use the saver to restore variables from disk
         # new_graph1 = tf.Graph()
         with tf.Session() as sess:
         #     sess.run(tf.global_variables_initializer())
         #     saver = tf.train.import_meta_graph('./lenet.meta')
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             print("Model restored.")
             output = tf.argmax(logits, 1)
             output = sess.run(output, feed_dict={x: X_images})


         fig, axs = plt.subplots(1,5, figsize=(16, 8))
         fig.subplots_adjust(hspace = .8, wspace=0.1)
         axs = axs.ravel()
         num_images = 5
         for i in range(num_images):
             axs[i].imshow(X_images[i].reshape(32,32))
```

```
            axs[i].set_title(('Prediction:',output[i],'True:',y_images[i]))
```

```
Model restored.
```



2.1.3 Analyze Performance

```
In [20]: ### Calculate the accuracy for these 9 new images.
         ### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate o

         # Load the model
         with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             print ("accuracy", sess.run(accuracy_operation, feed_dict={x: X_images, y: y_images
```

```
accuracy 0.0
```

2.1.4 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` 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:

```
In [21]: ### Print out the top five softmax probabilities for the predictions on the German traf
         ### Feel free to use as many code cells as needed.
         probabilities = []
         with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             probs = tf.nn.top_k(tf.nn.softmax(logits), k = 6)
```

```
            probabilities = sess.run(probs,feed_dict={x: X_images})
            print (probabilities)
        # fig, axs = plt.subplots(1,5, figsize=(15, 8))
        # fig.subplots_adjust(hspace = .8, wspace=2)
        # axs = axs.ravel()
        # for i in range(num_images):
        #     axs[i].imshow(X_images[i].reshape(32,32))
        #     axs[i].set_title(('Prediction:',output[i],'True:',y_images))
```

```
TopKV2(values=array([[  8.63556623e-01,   1.27355650e-01,   6.70961943e-03,
          1.39858213e-03,   4.89727536e-04,   2.01340896e-04],
       [  6.38022542e-01,   3.56648445e-01,   3.93287977e-03,
          4.01037425e-04,   3.96111631e-04,   3.74103984e-04],
       [  8.91324699e-01,   1.00955203e-01,   6.48887362e-03,
          6.35473174e-04,   3.86551517e-04,   6.65775806e-05],
       [  9.20284510e-01,   7.13649392e-02,   7.26093352e-03,
          5.57681080e-04,   2.60142057e-04,   1.01262864e-04],
       [  9.57505703e-01,   3.51276994e-02,   6.45353645e-03,
          4.28569037e-04,   1.97479705e-04,   1.48004663e-04]], dtype=float32), indices=array([[
       [18,  3,  5, 13, 11, 32],
       [ 3, 18,  5, 32, 13, 11],
       [ 3, 18,  5, 13, 32, 11],
       [ 3, 18,  5, 13, 32, 11]], dtype=int32))
```

2.1.5 Project Writeup

Once you have completed the code implementation, document your results in a project writeup using this template as a guide. The writeup can be in a markdown or pdf file.

2.2 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.

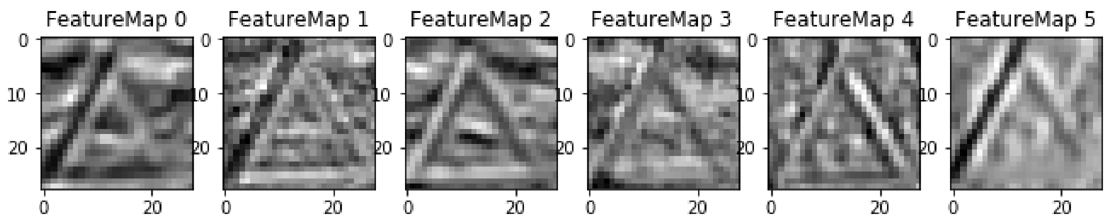
Provided for you below is the function code that allows you to get the visualization output of any tensorflow weight layer you want. The inputs to the function should be a stimuli image, one used during training or a new one you provided, and then the tensorflow variable name that represents the layer's state during the training process, for instance if you wanted to see what the LeNet lab's 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 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.

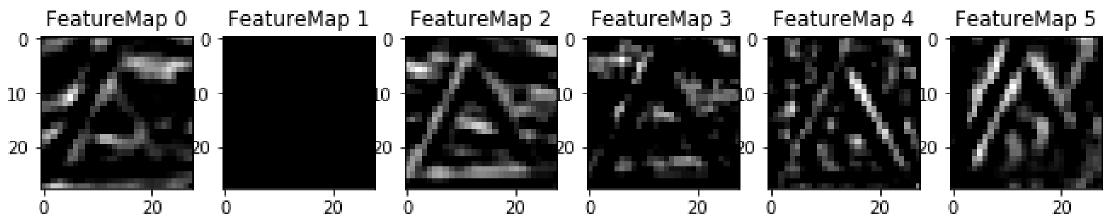
Your output should look something like this (above)

```
In [22]: ### 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
         # activation_min/max: can be used to view the activation contrast in more detail, by de
         # plt_num: used to plot out multiple different weight feature map sets on the same bloc

         def outputFeatureMap(image_input, tf_activation, activation_min=-1, activation_max=-1 ,
             # 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 var
             # If you get an error tf_activation is not defined it may be having trouble accessi
             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 eac
                 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 =ac
                 elif activation_max != -1:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmax=act
                 elif activation_min !=-1:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", vmin=act
                 else:
                     plt.imshow(activation[0,:,:, featuremap], interpolation="nearest", cmap="gr
```

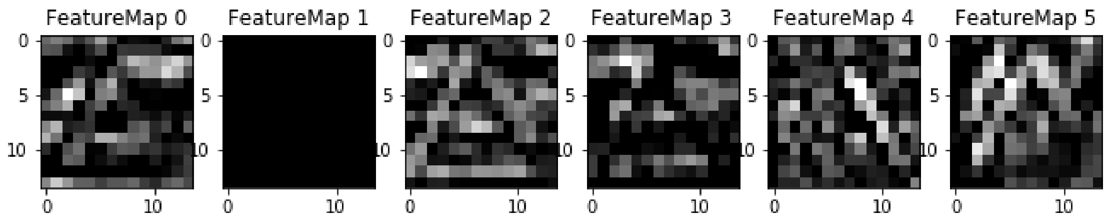
```
In [23]: # Load the model
         import tensorflow as tf
         with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1 = sess.graph.get_tensor_by_name('conv1:0')
             outputFeatureMap(X_train, conv1)
```



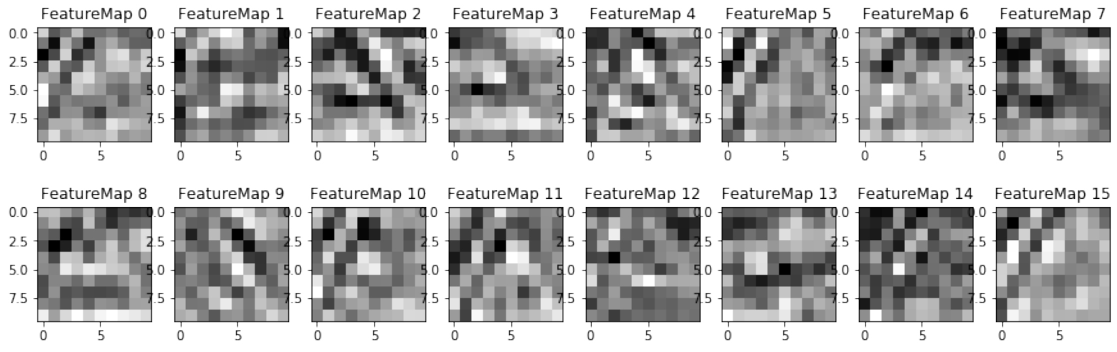
```
In [24]: with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1_relu = sess.graph.get_tensor_by_name('conv1_relu:0')
             outputFeatureMap(X_train, conv1_relu)
```



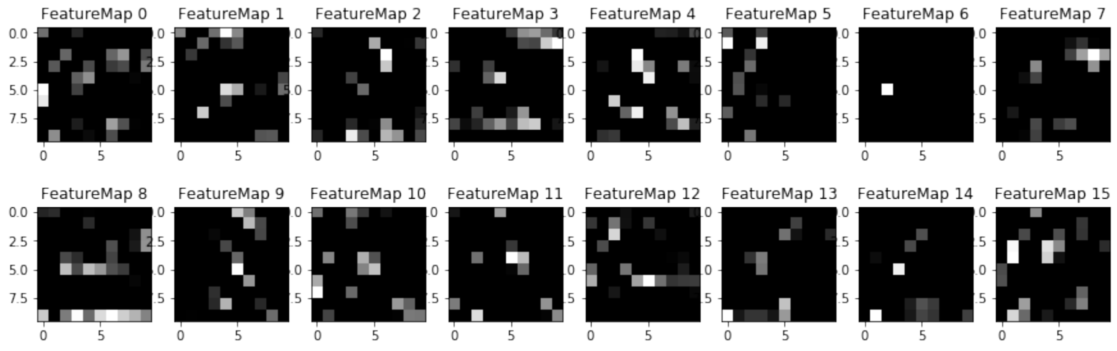
```
In [25]: with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1_pool = sess.graph.get_tensor_by_name('conv1_maxpool:0')
             outputFeatureMap(X_train, conv1_pool)
```



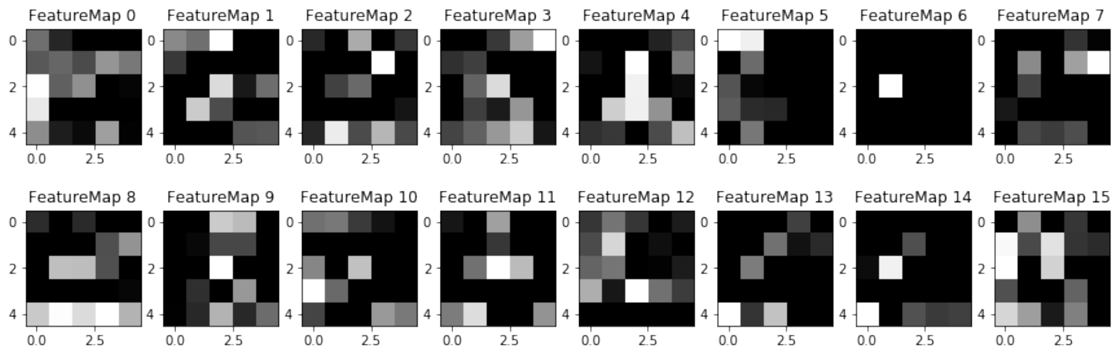
```
In [26]: with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1 = sess.graph.get_tensor_by_name('conv2:0')
             outputFeatureMap(X_train, conv1)
```



```
In [27]: with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1_relu = sess.graph.get_tensor_by_name('conv2_relu:0')
             outputFeatureMap(X_train, conv1_relu)
```



```
In [28]: with tf.Session() as sess:
             saver.restore(sess, tf.train.latest_checkpoint('.'))
             conv1_pool = sess.graph.get_tensor_by_name('conv2_maxpool:0')
             outputFeatureMap(X_train, conv1_pool)
```


FeatureMap 0
FeatureMap 1
FeatureMap 2
FeatureMap 3
FeatureMap 4
FeatureMap 5
FeatureMap 6
FeatureMap 7
FeatureMap 8
FeatureMap 9
FeatureMap 10
FeatureMap 11
FeatureMap 12
FeatureMap 13
FeatureMap 14
FeatureMap 15