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]:

```
# Load pickled data
import pickle
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import cv2
```

In [2]:

```
# Load training, validation and test
# Original data at: http://benchmark.ini.rub.de/?section=gtsrb&subsection=datase
t

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)](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 [3]:

```
print(np.unique(y_valid))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 2
2 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42]
```

In [79]:

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

# Number of training examples
n_train = X_train.shape[0]

# Number of validation examples
n_validation = X_valid.shape[0]

# Number of testing examples.
n_test = X_test.shape[0]

# What's the shape of an traffic sign image?
image_shape = X_train.shape[1:]

# How many unique classes/labels there are in the dataset.
labels = np.unique(y_valid)
n_classes = len(labels)

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 [5]:

```
import csv
# Load Sign Names
signnames =  './signnames.csv'
with open(signnames, mode='r') as infile:
    reader = csv.reader(infile)
    labels_names = {rows[0]:rows[1] for rows in reader}
print(labels_names['10'])
```

```
No passing for vehicles over 3.5 metric tons
```

In [6]:

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

num_images_disp = 4
fig, axes = plt.subplots(num_images_disp, num_images_disp,\
                         figsize=(15,15))
for ax in axes.flat:
    index = np.random.randint(0,n_train-1)
    ax.imshow(X_train[index])
    ax.title.set_text(labels_names[str(y_train[index])])
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
fig.subplots_adjust(hspace=0.5)
```

In [7]:

```
# Find one image for each label
fig, axes = plt.subplots(7, 7,\
                         figsize=(10,10))
y_train_list = list(y_train)
for label in labels:
    ii = label // 7
    jj = (label - ii*7)
    ax = axes[ii,jj]
    img_idx = y_train_list.index(label)
    ax.imshow(X_train[img_idx])
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.title.set_text(label)
```


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
20
STOP
1.0
0.5
0.0
0.0
0.5
1.0

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 [8]:

```
X_list = list(X_train.shape[:4])
X_list[3] = 1
print((X_list))
```

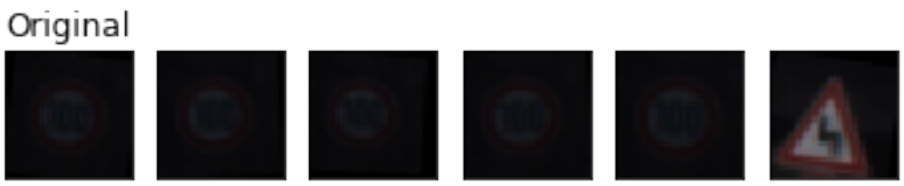
```
[34799, 32, 32, 1]
```

In [9]:

```
### Preprocess the data
def preprocImages(imgs, eq_hist = True):
    # Convert to Greyscale
    new_size = list(imgs.shape)
    new_size[3] = 1
    X_grey = np.zeros(tuple(new_size))

    for ii in range(new_size[0]):
        img = imgs[ii,:,:,:]
        img_grey = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY, dstCn = 4)
        if eq_hist:
            img_grey = cv2.equalizeHist(np.uint8(img_grey))
        X_grey[ii,:,:,0] = img_grey/255 - 0.5
    return X_grey

def jitterImages(imgs, Ngen = 3, delta_scale = (0.9,1.1), delta_angle = 10):
    # For each image, generate 'Ngen' randomly sampled images with:
    # - Global Scale with range delta_scale.
    # - Rotation with abs mangitude delta_angle (Degrees)
    # Random Translation is added by random selection of the center of rotation.
    def jitterImage(img, delta_scale = (0.9,1.1), delta_angle = 15):
        # Apply a random jitter.
        # Random Scale
        scale_range = delta_scale[1] - delta_scale[0]
        scale = delta_scale[0] + scale_range*np.random.rand()
        # Random Angle
        angle = 2*delta_angle*np.random.rand() - delta_angle
        # Random Center
        center = (np.random.randint(img.shape[0]),np.random.randint(img.shape[1]
))

        # Get Rotation
        Rmat = cv2.getRotationMatrix2D(center, angle, scale)

        # Apply Rotation
        return cv2.warpAffine(img, Rmat, dsize = (32,32))

    new_imgs = []
    for img in imgs:
        for n in range(Ngen):
            new_imgs.append(jitterImage(img, delta_scale, delta_angle))
    return np.array(new_imgs)

# Test Data Augmentation
```

```python
# Test Data Augmentation
Num_test = 5

test_idx = np.random.randint(low = 0, high = len(X_train), size = Num_test)
X_gen = jitterImages(X_train[test_idx], Ngen = Num_test)
y_gen = np.repeat(y_train[test_idx], Num_test)

# plot old image
ax =plt.subplot(1,6,1)
plt.imshow(X_train[0])
plt.title('Original')
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
# plot new images
for subplt in range(6):
    ax = plt.subplot(1,6,subplt + 1)
    plt.imshow(X_gen[subplt])
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

# Check Labels
print('OLD LABELS: ', y_train[test_idx])
print('NEW LABELS: ', y_gen)
print('Gen size: ', X_gen.shape)
```

```
OLD LABELS:  [ 7 21 13 33 10]
NEW LABELS:  [ 7  7  7  7  7 21 21 21 21 21 13 13 13 13 13 33 33 33
 33 33 10 10 10 10 10]
Gen size:  (25, 32, 32, 3)
```



In [83]:

```
# Augment training set with Jittered images
N_augment = 5
augmented_data_file = './traffic-signs-data/augmented_train_%d.p'%(N_augment)
gen_aug_data = False

if gen_aug_data:
    aug = {}
    aug['features'] = jitterImages(X_train[:], Ngen = N_augment)
    aug['labels'] = np.repeat(y_train[:], N_augment)
    with open(augmented_data_file, mode='wb') as f:
        pickle.dump(aug,f)
else:
    with open(augmented_data_file, mode='rb') as f:
        aug = pickle.load(f)

# Augmented Data vector
X_aug, y_aug = aug['features'], aug['labels']
print(y_aug)
n_aug = X_aug.shape[0]
print('Number of Augmented images: ' +  str(n_aug))

# plot old image
ax =plt.subplot(1,6,1)
plt.imshow(X_train[0])
plt.title('Original')
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
# plot new images
for subplt in range(6):
    ax = plt.subplot(1,6,subplt + 1)
    plt.imshow(X_aug[subplt])
    plt.title('Label: %d'%(y_aug[subplt]))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

# Append data
X_train_aug = np.concatenate((X_train, X_aug), axis = 0)
y_train_aug = np.concatenate((y_train, y_aug), axis = 0)
```

```
[41 41 41 ..., 25 25 25]
Number of Augmented images: 173995
```

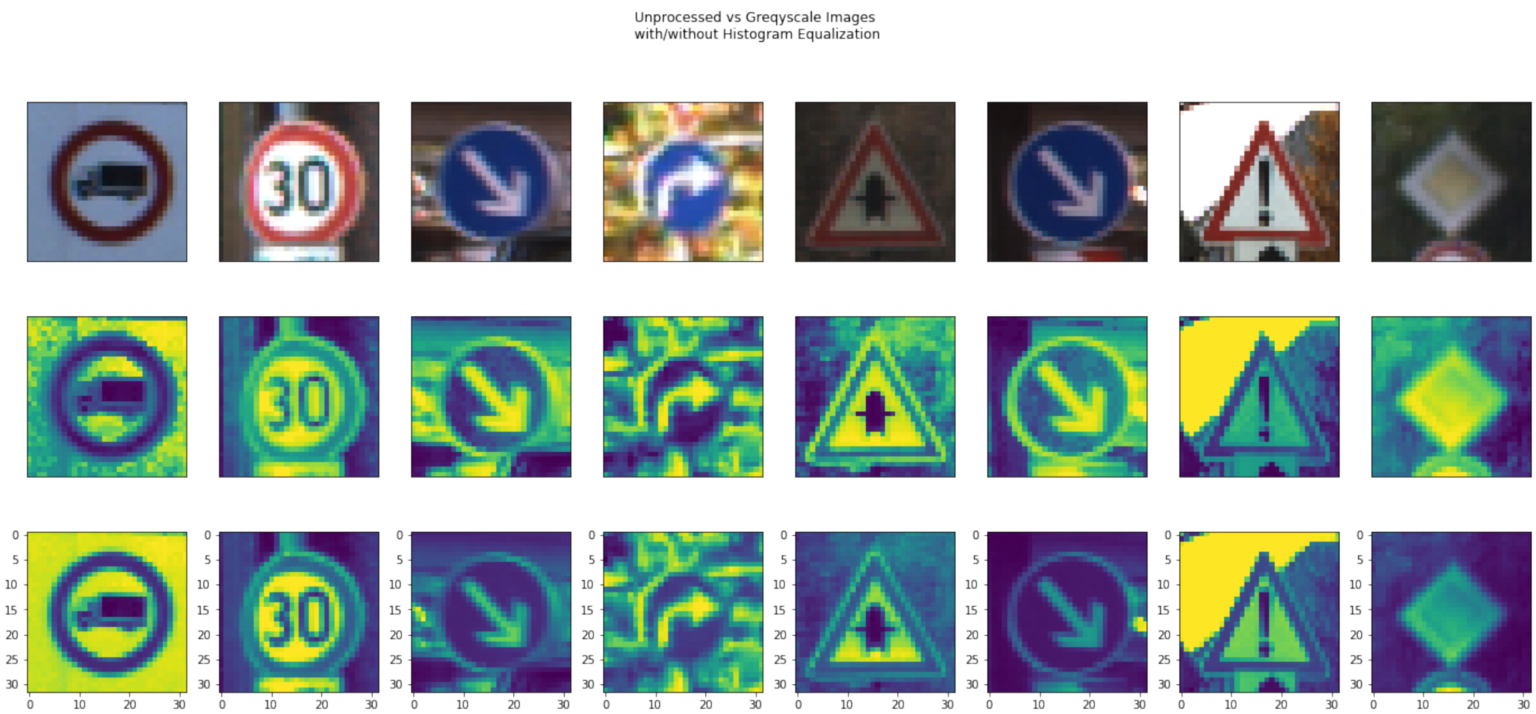
In [11]:

```
# Test Pre-Processing Pipeline
num_preproc_test = (8,3)
num_test = np.prod(num_preproc_test)
print(X_test.shape)
X_eq_off = preprocImages(X_test[:num_preproc_test[0]], eq_hist = False)
X_eq_on = preprocImages(X_test[:num_preproc_test[0]], eq_hist = True)
print(X_eq_on.shape)


fig, axes = plt.subplots(num_preproc_test[1], num_preproc_test[0],\
                         squeeze=True, figsize=(24,10))


for ii in range(num_preproc_test[0]):
    axes[0,ii].imshow(X_test[ii])
    axes[1,ii].imshow(X_eq_on[ii].squeeze())
    axes[2,ii].imshow(X_eq_off[ii].squeeze())
    for jj in range(2):
        axes[jj,ii].get_xaxis().set_visible(False)
        axes[jj,ii].get_yaxis().set_visible(False)
fig.suptitle('Unprocessed vs Greqyscale Images\n with/without Histogram Equaliza
tion')
```

```
(12630, 32, 32, 3)
(8, 32, 32, 1)
```

Out[11]:

```
<matplotlib.text.Text at 0x7f85b373a860>
```



In [12]:

```
from sklearn.utils import shuffle

# Perform Pre-processing
X_valid_proc = preprocImages(X_valid[:], eq_hist = True)
y_valid_proc = y_valid
X_test = preprocImages(X_test[:], eq_hist = True)
y_test_proc = y_test

# Pre-process and Shuffle
X_train_proc = preprocImages(X_train_aug[:], eq_hist = True)
X_train_proc, y_train_proc = shuffle(X_train_proc, y_train_aug)
```

In [13]:

```
print(X_train_aug.shape)
```

```
(208794, 32, 32, 3)
```

Model Architecture

In [75]:

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

### Define your architecture here.
class signRecognizer:
    # Input data
    x = None
    y = None
    one_hot_y = None
    keep_prob = None

    # Initialization
    mu_def = 0
    sigma_def = 0.1

    # Network Layers
    network = {}
    weights = {}
    biases = {}

    # Optimizer
    learning_rate = 0.0002

    # Evaluation
    saver = None
    accuracy_operation = None
    loss_operation = None
    training_op = None
```

```
    training_op = None

    def __init__(self):
        # Setup Object

        # INput and Output Placeholders
        self.x = tf.placeholder(tf.float32, (None, image_shape[0], image_shape[1], 1))
        self.y = tf.placeholder(tf.int32, (None))
        self.one_hot_y = tf.one_hot(self.y, n_classes)

        # Dropout
        self.keep_prob = tf.placeholder(tf.float32,None)

        # Variable
        self.weights = {'w1': self.getWeights([5,5,1,24]),\
                        'w2':  self.getWeights([5,5,24,32]),\
                        'w3':  self.getWeights([3,3,32,64]),\
                        'wfc1': self.getWeights([576,128]),\
                        'wfc2': self.getWeights([128,n_classes])}

        self.bias = {'b1': tf.zeros([24]),\
                    'b2': tf.zeros([32]),\
                    'b3': tf.zeros([64]),\
                    'bfc1': tf.zeros([128]),\
                    'bfc2': tf.zeros([n_classes])}

        self.saver = tf.train.Saver(max_to_keep = 10)

    def getIO(self):
        return self.x, self.y, self.keep_prob

    def getWeights(self,shape):
        return tf.Variable(tf.truncated_normal(shape,\
                                    mean= self.mu_def,\
                                    stddev=self.sigma_def,\
                                    dtype=tf.float32))

    def getConv2Dlayer(self, x, weights, bias, stride = 2):
        # Define a Convolutional Layer with Max Pooling.
        conv_layer = tf.nn.conv2d(input = x,\
                            filter = weights,\
                            strides = [1,1,1,1],\
                            padding = 'VALID')
        conv_layer = tf.nn.bias_add(conv_layer, bias)

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

        # Pooling.
        conv_layer = tf.nn.max_pool(conv_layer,\
                               ksize = [1,stride,stride,1],\
                               strides = [1,stride,stride,1],\
                               padding = 'VALID')
```

```
        conv_layer = tf.nn.dropout(conv_layer, self.keep_prob)
        return conv_layer

    def getFClayer(self, x, weights, bias, activation = True):
        # Define a fully connected layer.
        fc = tf.nn.bias_add(tf.matmul(x, weights), bias)
        if activation:
            # Activation.
            return tf.nn.dropout(tf.nn.relu(fc), self.keep_prob)
        else:
            return tf.nn.dropout(fc, self.keep_prob)

    def initNetwork(self):
        # Initalize DNN

        # 2D Conv: (32,32,1) -> (14,14,24) [5 px]
        self.network['conv1'] = self.getConv2Dlayer(self.x,self.weights['w1'],\
                                                    self.bias['b1'])
        print(self.network['conv1'].get_shape())
        # 2D Conv: (14, 14, 24) -> (5, 5, 32) [5 px]
        self.network['conv2'] = self.getConv2Dlayer(self.network['conv1'],\
                                                    self.weights['w2'],\
                                                    self.bias['b2'])
        print(self.network['conv2'].get_shape())
        # 2D Conv: (5, 5, 32) -> (3,3,64) [3 px]
        self.network['conv3'] = self.getConv2Dlayer(self.network['conv2'],\
                                                    self.weights['w3'],\
                                                    self.bias['b3'],\
                                                    stride = 1)
        print(self.network['conv3'].get_shape())

#         # 1D Conv: (3,3,128) -> (3,3,32)
#         self.network['conv4'] = self.getConv2Dlayer(self.network['conv3'],\
#                                                     self.weights['w4'],\
#                                                     self.bias['b4'],\
#                                                     stride = 1)
#         print(self.network['conv4'].get_shape())

        # Flatten. (3,3,32) - >(576)
        self.network['flat'] = tf.reshape(self.network['conv3'],\
                                          shape = [-1,576])

        # Layer 3: Fully Connected. Input = 576. Output = 120.
        self.network['fc1'] = self.getFClayer(self.network['flat'],\
                                              self.weights['wfc1'],\
                                              self.bias['bfc1'],\
                                              activation = True)

        # Layer 6: Fully Connected. Input = 120. Output = n_classes.
        self.network['logits'] = self.getFClayer(self.network['fc1'],\
                                                 self.weights['wfc2'],\
                                                 self.bias['bfc2'],\
```

```
                                                activation = False)


        self.logits = self.network['logits']

        # Define Optimization
        self.setupOptimization()
        self.setEval()

    def setupOptimization(self):
        self.setLoss()
        self.setOptimizer()

    def setLoss(self):
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=self.one_
hot_y,\
                                                                logits=self.logi
ts)
        self.loss_operation = tf.reduce_mean(cross_entropy)

    def setOptimizer(self):
        self.optimizer = tf.train.AdamOptimizer(learning_rate = self.learning_ra
te)
        self.training_op = self.optimizer.minimize(self.loss_operation)

    def setEval(self):
        correct_prediction = tf.equal(tf.argmax(self.logits, 1), tf.argmax(self.
one_hot_y, 1))
        self.accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.
float32))
```

In [76]:

```
# Tensorflow meta-parameters
EPOCHS = 30
BATCH_SIZE = 128

N = signRecognizer()
N.initNetwork()
x,y,kp = N.getIO()
```

```
(?, 14, 14, 24)
(?, 5, 5, 32)
(?, 3, 3, 64)
```

In [77]:

```
# Define the Evaluation function for Cross-Validation and Testing
def evaluate(X_data, y_data, sess):
    num_examples = len(X_data)
    total_accuracy = 0
    for offset in range(0, num_examples, BATCH_SIZE):
        batch_x, batch_y = X_data[offset:offset+BATCH_SIZE], y_data[offset:offse
t+BATCH_SIZE]
        accuracy, loss = sess.run([N.accuracy_operation, N.loss_operation],\
                                  feed_dict={x: batch_x, y: batch_y, kp: 1.})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples, loss
```

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 [78]:

```
### Train your model here.
accuracy = []
loss = []

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

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train_proc, y_train_proc = shuffle(X_train_proc, y_train_proc)

        # Batches
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train_proc[offset:end], y_train_proc[offset:end
]
            sess.run(N.training_op, feed_dict={x: batch_x,\
                                                y: batch_y,\
                                                kp: 0.5})

        validation_accuracy, validation_loss = evaluate(X_valid_proc, y_valid_pr
oc, sess)
        print("EPOCH {} ...".format(i+1))
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print("Validation Loss = {:3.3f}".format(validation_loss))
        accuracy.append(validation_accuracy)
        loss.append(validation_loss)
        print()

        N.saver.save(sess, './signClass.ckpt')
    print("Model saved")

### 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.
```

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.229
Validation Loss = 3.543

EPOCH 2 ...
Validation Accuracy = 0.478
Validation Loss = 2.981

EPOCH 3 ...
Validation Accuracy = 0.571
```

```
Validation Loss = 2.491

EPOCH 4 ...
Validation Accuracy = 0.664
Validation Loss = 1.914

EPOCH 5 ...
Validation Accuracy = 0.697
Validation Loss = 1.135

EPOCH 6 ...
Validation Accuracy = 0.743
Validation Loss = 0.832

EPOCH 7 ...
Validation Accuracy = 0.787
Validation Loss = 0.740

EPOCH 8 ...
Validation Accuracy = 0.810
Validation Loss = 0.513

EPOCH 9 ...
Validation Accuracy = 0.835
Validation Loss = 0.350

EPOCH 10 ...
Validation Accuracy = 0.844
Validation Loss = 0.376

EPOCH 11 ...
Validation Accuracy = 0.859
Validation Loss = 0.305

EPOCH 12 ...
Validation Accuracy = 0.879
Validation Loss = 0.306

EPOCH 13 ...
Validation Accuracy = 0.888
Validation Loss = 0.203

EPOCH 14 ...
Validation Accuracy = 0.898
Validation Loss = 0.184

EPOCH 15 ...
Validation Accuracy = 0.907
Validation Loss = 0.189

EPOCH 16 ...
Validation Accuracy = 0.918
Validation Loss = 0.155
```

```
EPOCH 17 ...
Validation Accuracy = 0.920
Validation Loss = 0.168

EPOCH 18 ...
Validation Accuracy = 0.931
Validation Loss = 0.146

EPOCH 19 ...
Validation Accuracy = 0.933
Validation Loss = 0.113

EPOCH 20 ...
Validation Accuracy = 0.933
Validation Loss = 0.116

EPOCH 21 ...
Validation Accuracy = 0.934
Validation Loss = 0.100

EPOCH 22 ...
Validation Accuracy = 0.941
Validation Loss = 0.075

EPOCH 23 ...
Validation Accuracy = 0.944
Validation Loss = 0.075

EPOCH 24 ...
Validation Accuracy = 0.953
Validation Loss = 0.049

EPOCH 25 ...
Validation Accuracy = 0.954
Validation Loss = 0.080

EPOCH 26 ...
Validation Accuracy = 0.953
Validation Loss = 0.061

EPOCH 27 ...
Validation Accuracy = 0.955
Validation Loss = 0.042

EPOCH 28 ...
Validation Accuracy = 0.958
Validation Loss = 0.066

EPOCH 29 ...
Validation Accuracy = 0.957
Validation Loss = 0.048
```

```
EPOCH 30 ...
Validation Accuracy = 0.961
Validation Loss = 0.044

Model saved
```

In [87]:

```python
# Calculate Final Train, Validation and Test accuracies

with tf.Session() as sess:
    N.saver.restore(sess, "./signClass.ckpt")

    # Training Accuracy
    # Batches
    batch_accuracy = []
    batch_loss = []
    for offset in range(0, num_examples, BATCH_SIZE):
        end = offset + BATCH_SIZE
        batch_x, batch_y = X_train_proc[offset:end], y_train_proc[offset:end]
        batch_i_accuracy, batch_i_loss = evaluate(batch_x, batch_y, sess)
        batch_accuracy.append(batch_i_accuracy)
        batch_loss.append(batch_i_loss)
    batch_accuracy = np.array(batch_accuracy)
    batch_loss = np.array(batch_loss)
    print("Train Accuracy = {:.3f}".format(np.mean(batch_accuracy)))
    print("Train Loss = {:3.3f}".format(np.mean(batch_loss)))

    # Validation Set
    validation_accuracy, validation_loss = evaluate(X_valid_proc, y_valid_proc,
sess)
    print("Validation Accuracy = {:.3f}".format(validation_accuracy))
    print("Validation Loss = {:3.3f}".format(validation_loss))

    # Validation Set
    test_accuracy, test_loss = evaluate(X_test, y_test, sess)
    print("Test Accuracy = {:.3f}".format(test_accuracy))
    print("Test Loss = {:3.3f}".format(test_loss))
```

```
INFO:tensorflow:Restoring parameters from ./signClass.ckpt
Train Accuracy = 0.945
Train Loss = 0.421
Validation Accuracy = 0.961
Validation Loss = 0.044
Test Accuracy = 0.938
Test Loss = 0.369
```

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.

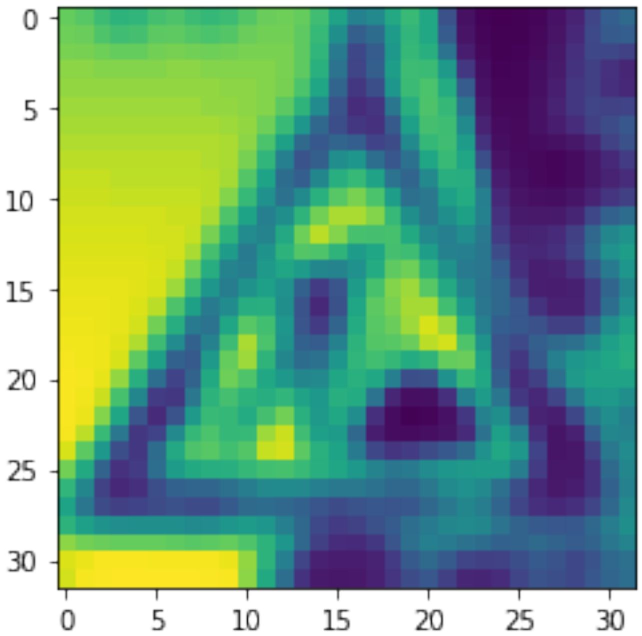
You may find `signnames.csv` useful as it contains mappings from the class id (integer) to the actual sign name.

Load and Output the Images

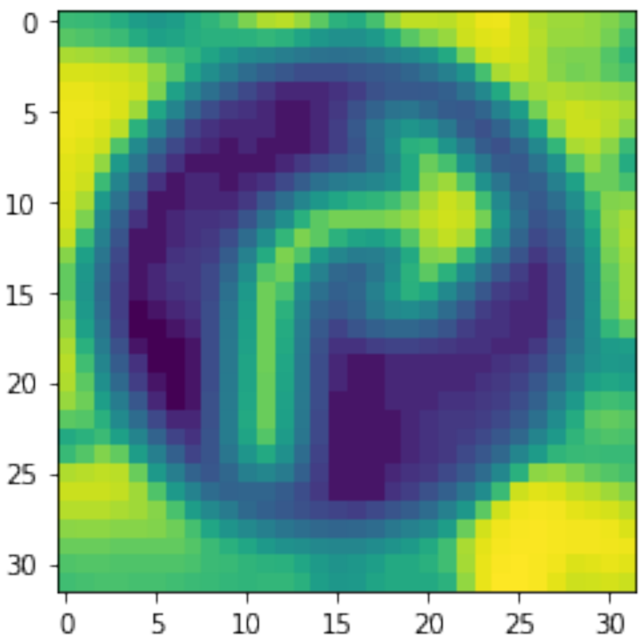
In [42]:

```
import os
import cv2
#Load new images and resize
new_signs_dir = './sign_images/'
filenames = os.listdir(new_signs_dir)
test_images = []
test_labels = []

for file in filenames:
    if file.endswith('small.png'):
        # load file
        img = cv2.cvtColor(cv2.imread(new_signs_dir+file),cv2.COLOR_BGR2RGB)
        img = cv2.GaussianBlur(img,(35,35),0)
        img = cv2.resize(img,(32,32),cv2.INTER_AREA)
        test_images.append(np.array(img))
        label = int(file[4:6])
        test_labels.append(label)

# Pre-Process
test_images = preprocImages(np.array(test_images))
for ii in range(len(test_images)):
    plt.imshow(test_images[ii,:,:,0])
    plt.title(test_labels[ii])
    plt.show()

# Example IMage for comparison
plt.imshow(X_valid_proc[0,:,:,0])
```

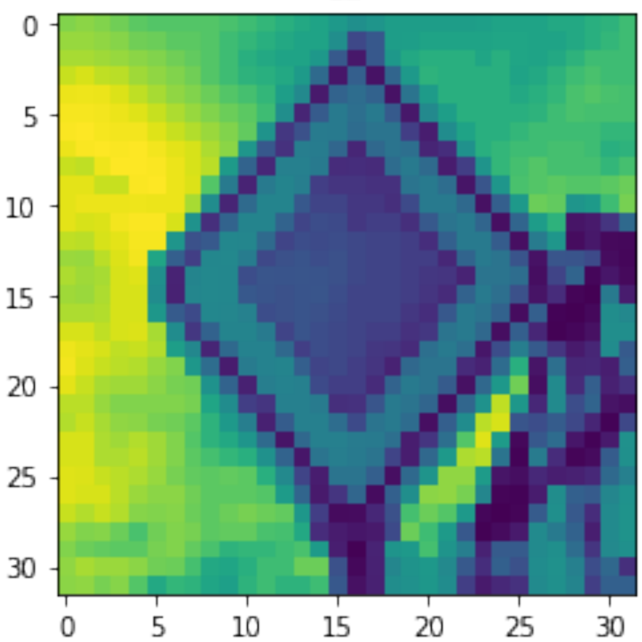
25

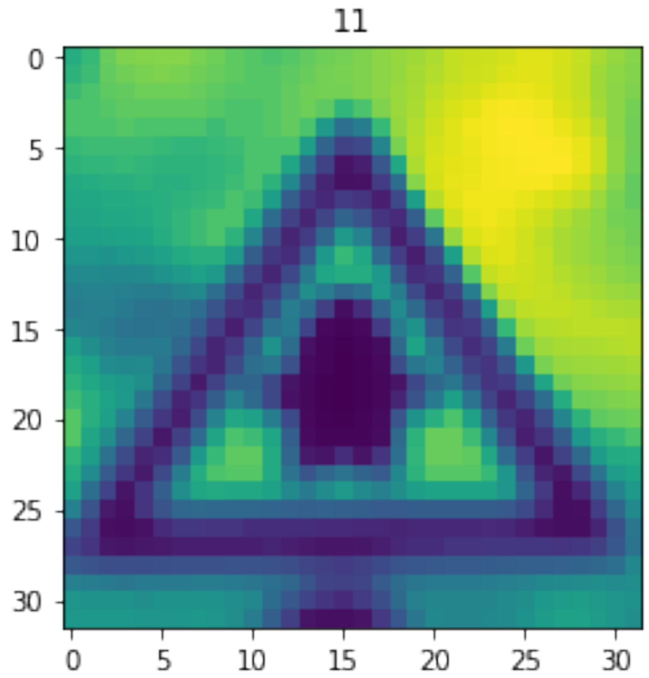


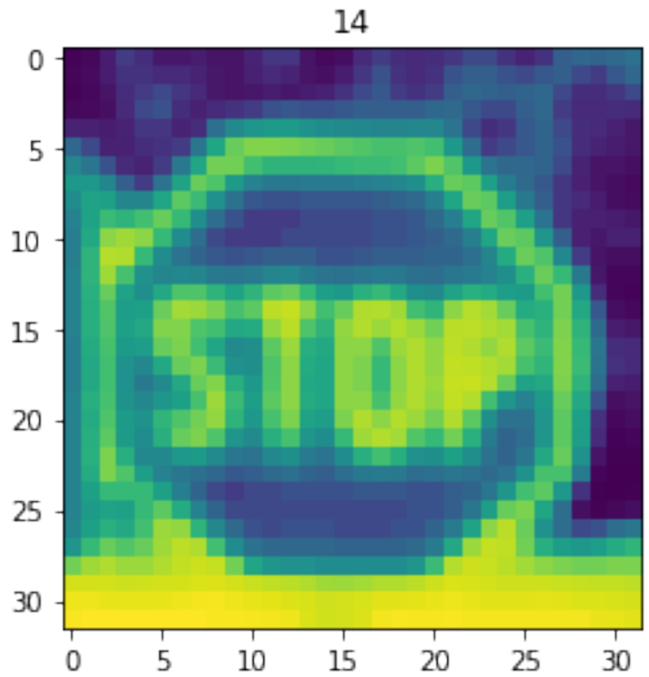
33

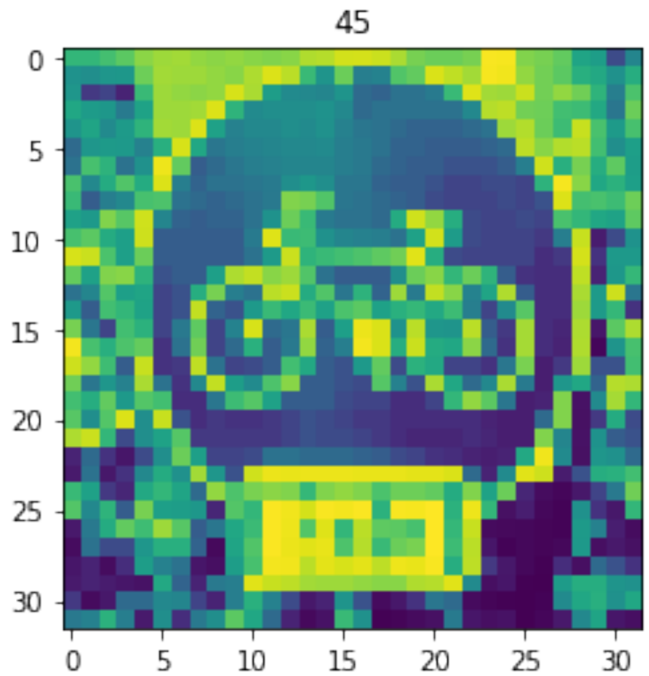


12



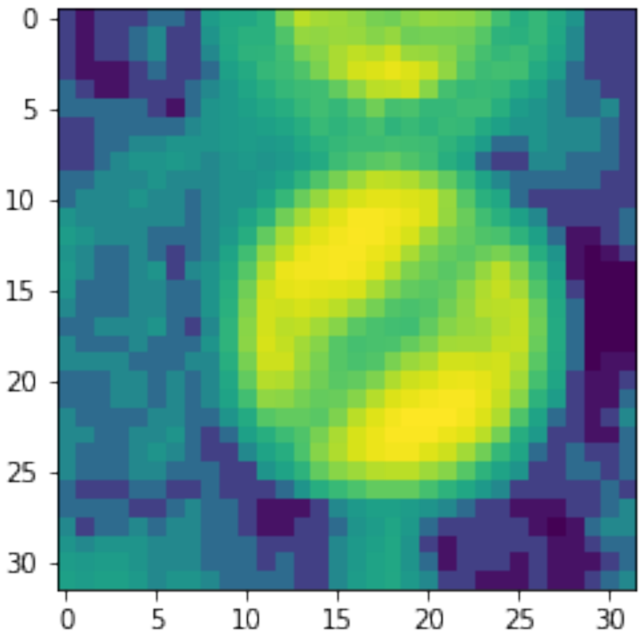






Out[42]:

```
<matplotlib.image.AxesImage at 0x7f84f1d8ec50>
```



Predict the Sign Type for Each Image

In [47]:

```
### Run the predictions here and use the model to output the prediction for each image.
y_new = []
nn_top_five = tf.nn.top_k(tf.nn.softmax(N.logits),5)
with tf.Session() as sess:
    N.saver.restore(sess, "./signClass.ckpt")
    test_img_logits, top_five = sess.run([N.logits, nn_top_five],\
                                            feed_dict={x: test_images, kp: 1.})

    predict = sess.run(tf.argmax(test_img_logits, axis=1))

# compare versus actual
print('Actual: ', test_labels)
print('Predict: ',predict)
# print(test_img_logits)
print('Predict == Actual: ', np.equal(predict, test_labels))
print('Accuracy = ', np.sum(np.equal(predict, test_labels))/len(test_images))
print('Top Five Predictions: ', top_five)
```

```
INFO:tensorflow:Restoring parameters from ./signClass.ckpt
Actual:  [25, 33, 12, 11, 14, 45]
Predict:  [25 33 12 11 14 14]
Predict == Actual:  [ True  True  True  True  True False]
Accuracy =  0.833333333333
Top Five Predictions:  TopKV2(values=array([[  9.97976243e-01,   4.8
8073245e-04,   3.76911252e-04,
          3.58037505e-04,   1.97129280e-04],
       [  9.98315930e-01,   5.09557838e-04,   3.08923802e-04,
          1.97715563e-04,   1.95844186e-04],
       [  9.99932051e-01,   4.28852218e-05,   6.58355202e-06,
          3.35734558e-06,   3.07143296e-06],
       [  9.43934679e-01,   5.00395410e-02,   1.90127012e-03,
          8.23199400e-04,   7.53696833e-04],
       [  9.99750435e-01,   4.93289444e-05,   3.52766074e-05,
          1.98026537e-05,   1.81333689e-05],
       [  5.48183098e-02,   4.28706072e-02,   4.20097038e-02,
          4.01671864e-02,   3.85503359e-02]], dtype=float32), indice
s=array([[25, 30, 11, 21, 29],
       [33, 35,  6, 39, 37],
       [12, 40, 38, 35, 32],
       [11, 30, 21, 28, 20],
       [14, 38, 17, 34,  8],
       [14, 40, 11,  1, 30]], dtype=int32))
```

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,  0.0789
3497,
         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:

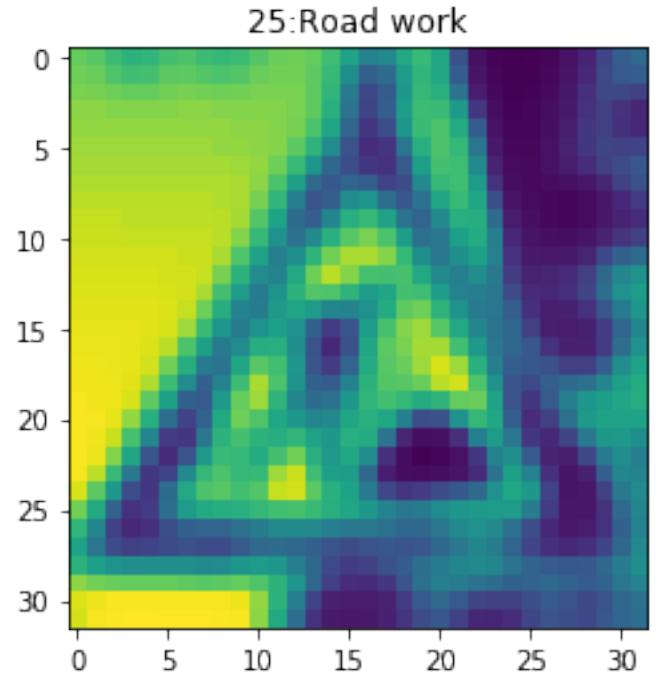
```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5]
,
       [0, 1,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

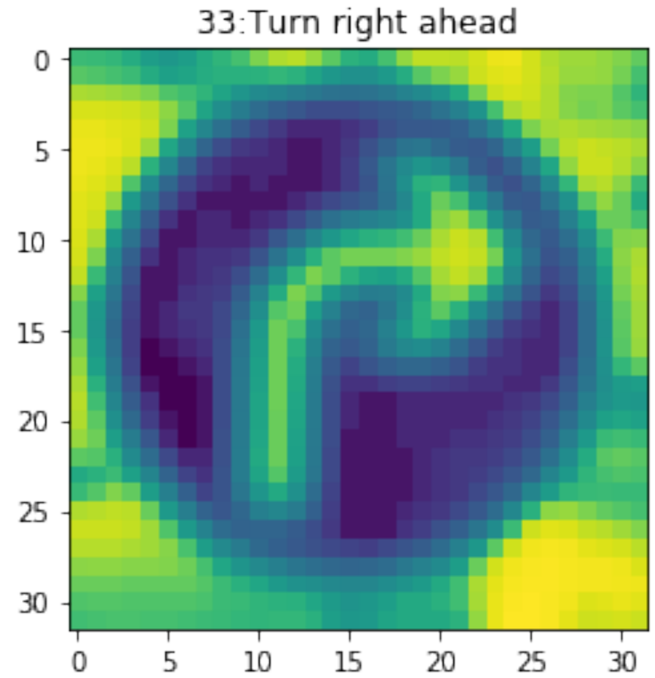
In [97]:

```
### Print out the top five softmax probabilities for the predictions on the Germ
an traffic sign images found on the web.
### Feel free to use as many code cells as needed.
for ii in range(5):
    print('Test image %d: %s (#%d)'%(ii,labels_names[str(test_labels[ii])],test_
labels[ii]))
    if top_five[1][ii][0] == int(test_labels[ii]):
        print('\tPrediction: Correct')
    else:
        print('\tPrediction: Incorrect')
    # print('Top Five Predictions: ', top_five[1][ii])
    print('Top Five Labels: ')
    for jj in range(5):
        print('\t %s (%2.2f)'%(labels_names[str(top_five[1][ii][jj])], 100*top_f
ive[0][ii][jj]))
    #print('\tTop Five Probabilities: ', top_five[0][ii])

    plt.imshow(test_images[ii,:,:,0])
    plt.title(str(test_labels[ii])+':' + labels_names[str(test_labels[ii])])
    plt.show()
```

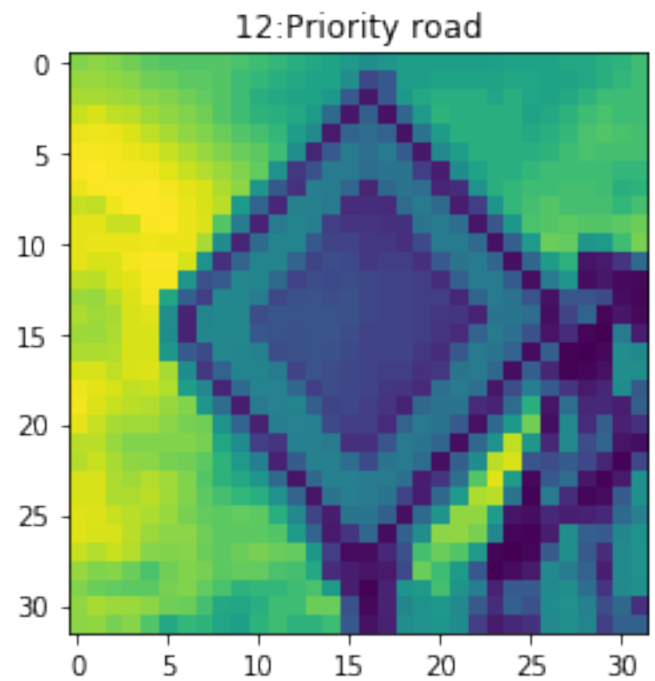
```
Test image 0: Road work (#25)
        Prediction: Correct
Top Five Labels:
         Road work (99.80)
         Beware of ice/snow (0.05)
         Right-of-way at the next intersection (0.04)
         Double curve (0.04)
         Bicycles crossing (0.02)
```



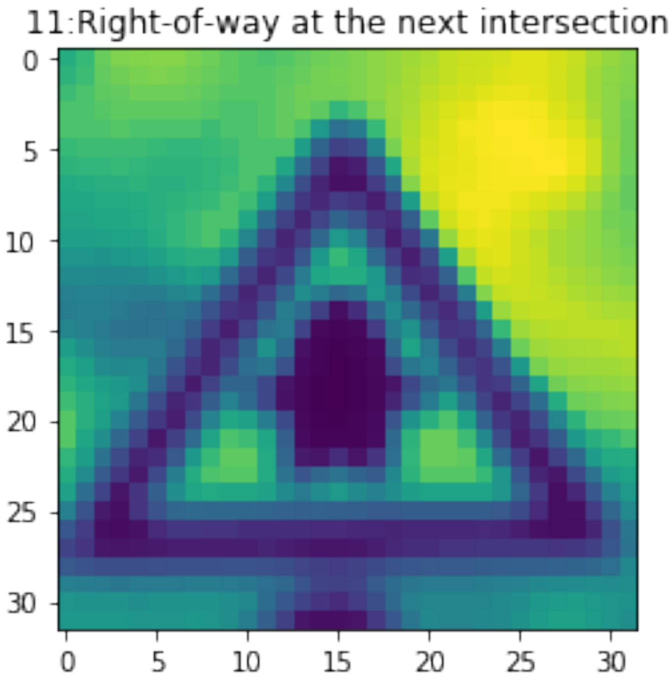
```
Test image 1: Turn right ahead (#33)
        Prediction: Correct
Top Five Labels:
         Turn right ahead (99.83)
         Ahead only (0.05)
         End of speed limit (80km/h) (0.03)
         Keep left (0.02)
         Go straight or left (0.02)
```



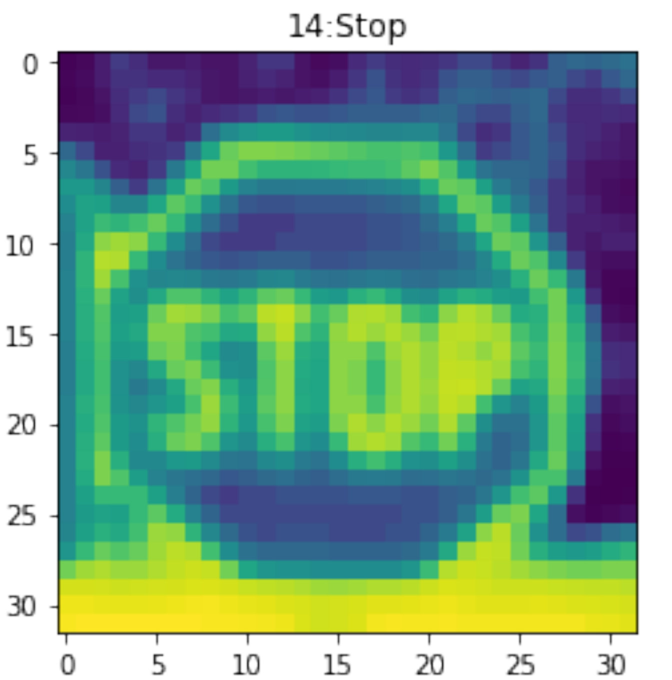

```
Test image 2: Priority road (#12)
        Prediction: Correct
Top Five Labels:
         Priority road (99.99)
         Roundabout mandatory (0.00)
         Keep right (0.00)
         Ahead only (0.00)
         End of all speed and passing limits (0.00)
```



```
Test image 3: Right-of-way at the next intersection (#11)
        Prediction: Correct
Top Five Labels:
         Right-of-way at the next intersection (94.39)
         Beware of ice/snow (5.00)
         Double curve (0.19)
         Children crossing (0.08)
         Dangerous curve to the right (0.08)
```




```
Test image 4: Stop (#14)
        Prediction: Correct
Top Five Labels:
         Stop (99.98)
         Keep right (0.00)
         No entry (0.00)
         Turn left ahead (0.00)
         Speed limit (120km/h) (0.00)
```



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.

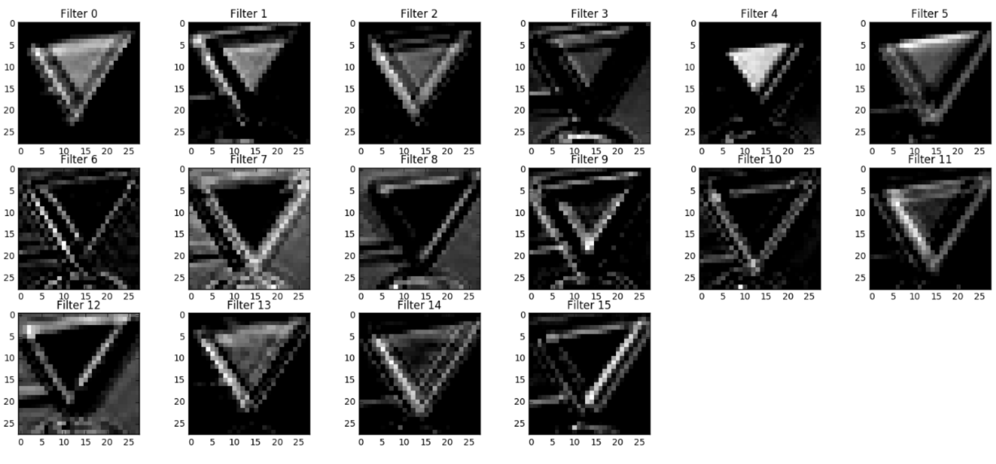
> **Note**: Once you have completed all of the code implementations and successfully answered each question above, you may finalize your work by exporting the iPython Notebook as an HTML document. You can do this by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. 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 feature maps
# tf_activation: should be a tf variable name used during your training procedure that represents the calculated state of a specific weight layer
# activation_min/max: can be used to view the activation contrast in more detail, by default matplot sets min and max to the actual min and max values of the 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 expects
    # with size, normalization, ect if needed
    # image_input =
    # Note: x should be the same name as your network's tensorflow data placeholder variable
    # If you get an error tf_activation is not defined it may be having trouble 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 show 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")
```