WRITEUP

This is the writeup file for the P2 project. I've based it on the jupyter notebook used to run the actual project, but have included the writeup in cells identified by the ## WRITEUP heading.

Step 0: Load The Data

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

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

training_file = 'traffic-signs-data/train.p'
testing_file = 'traffic-signs-data/test.p'

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

X_train, y_train = train['features'], train['labels']
X_test, y_test = test['features'], test['labels']
print(len(X_train))

assert len(test['features']) == len(test['labels'])
assert len(train['features']) == len(train['labels'])
```

```
39209
```

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 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.

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

WRITEUP

1. Provide a basic summary of the data set and identify where in your code the summary was done. In the code, the analysis should be done using python, numpy and/or pandas methods rather than hardcoding results manually.

The cell below calculates the necessary statistics. To get the number of examples I used a simple len() function. To get the shape of the image and number of unique classes, I used numpy libraries.

In [2]:
```python
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results

import numpy as np
import pandas as pd

# TODO: Number of training examples
n_train = len(train['features'])

# TODO: Number of testing examples.
n_test = len(test['features'])

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

# TODO: How many unique classes/labels there are in the dataset.
n_classes = len(np.unique(train['labels']))

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 = 39209
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 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.

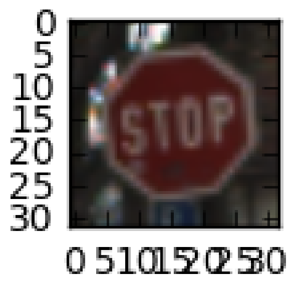
WRITEUP

1. Include an exploratory visualization of the dataset and identify where the code is in your code file.

The following few cells are devoted to exploratory visualization of the data. I've used matplotlib and random libraries to write a snipet of code that randomly picks an image from the set and displays it. Further below I've included a histogram of the data, to see how many examples of each class we have in the training set. We can observe some pretty uneven distribution, which probably corresponds to the probablility with which the signs were observed in real life. This is also done using matplotlib library.

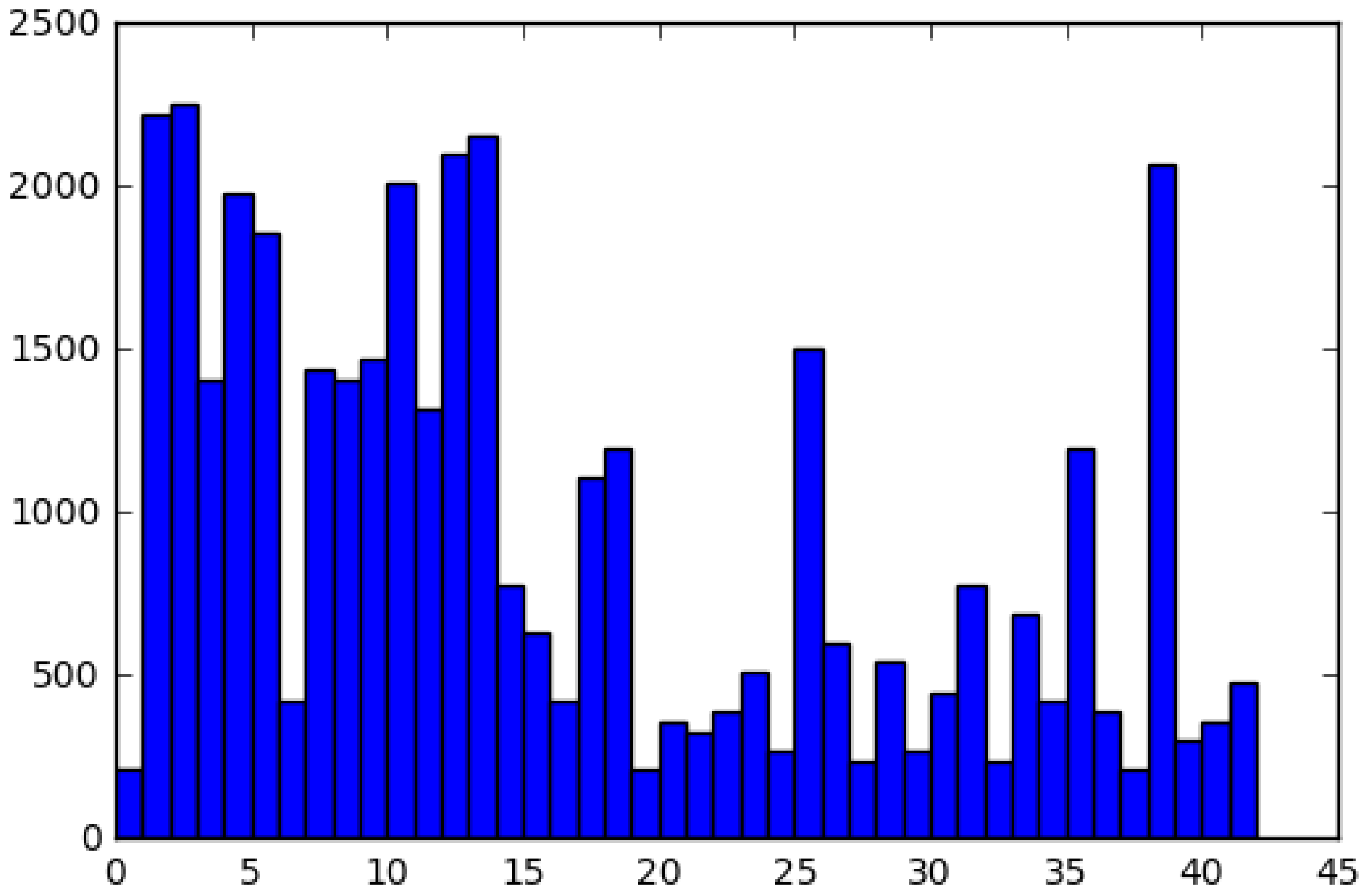
In [3]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import matplotlib.pyplot as plt
import random

# Visualizations will be shown in the notebook.
%matplotlib inline

index = random.randint(0, len((train['labels'])))
image = train['features'][index].squeeze()

plt.figure(figsize=(1,1))
plt.imshow(image)
```

Out[3]:
```
<matplotlib.image.AxesImage at 0x1234fcba8>
```



In [4]:
```
plt.hist(train['labels'],bins=range(len(np.unique(train['labels']))))
```

```
Out[4]: (array([  210.,  2220.,  2250.,  1410.,  1980.,  1860.,   420.,  1440.,
                1410.,  1470.,  2010.,  1320.,  2100.,  2160.,   780.,   630.,
                 420.,  1110.,  1200.,   210.,   360.,   330.,   390.,   510.,
                 270.,  1500.,   600.,   240.,   540.,   270.,   450.,   780.,
                 240.,   689.,   420.,  1200.,   390.,   210.,  2070.,   300.,
                 360.,   480.]),
         array([ 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]),
         <a list of 42 Patch objects>)
```



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.

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

- Neural network architecture
- 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.

NOTE: 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!

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.

WRITEUP

1. Describe how, and identify where in your code, you preprocessed the image data. What tecniques were chosen and why did you choose these techniques? Consider including images showing the output of each preprocessing technique. Pre-processing refers to techniques such as converting to grayscale, normalization, etc.

I've experimented with several techniques to preprocess the data, and settled for a simple normalization function that can be seen in the cell below. It will ensure feature values between 0.1 and o.9, removing any extreme values, and also ensuring the value is not zero. I wanted to build a network that works well with color images, as it seems that using the color information (instead of greyscale) more closely mimick how a human brain processes signs (color information can be important to correctly identify the sign). I decided against generating additional images for the classes with low frequency, because I wanted the probabilities of finding the sign in real life to be maintained, rather than artificially over-representing some signs that are rarely seen 'in the wild'. One step I tried to implement but did not get to work reliably was generating additional data by introducing a random rotation by +/-5 degrees and a slight shift in the image position. If implemented, I believe this would have improved the performance of my network further.

In [5]:
```
##TODO: normalize frequency of the images
##TODO: create jittered dataset - shift position and rotate angle / transform
, etc
##TODO: experiment with hue, contrast, etc.
##TODO: experiment with adjusting greyscale

def prep_img(img):

    return img / 255 * 0.8 + 0.1
```

In [6]:
```
##Image before processing
index = 333
image_pre = X_train[index]
```

WRITEUP

2. Describe how, and identify where in your code, you set up training, validation and testing data. How much data was in each set? Explain what techniques were used to split the data into these sets. (OPTIONAL: As described in the "Stand Out Suggestions" part of the rubric, if you generated additional data for training, describe why you decided to generate additional data, how you generated the data, identify where in your code, and provide example images of the additional data)

The code for splitting the data into training and validation sets is contained in the cell below. I used the scikit learn function that conveniently splits data into training and validation data sets for both features and labels, and also shuffles them. I've used the default setting of the function, which resulted in 75% of the data being alocated to the training, and 25% to validation - as that seemed like a sensible split.

My final training set had 29,406 number of images. My validation set and test set had 9,803 and 12,630 number of images respecitvely.

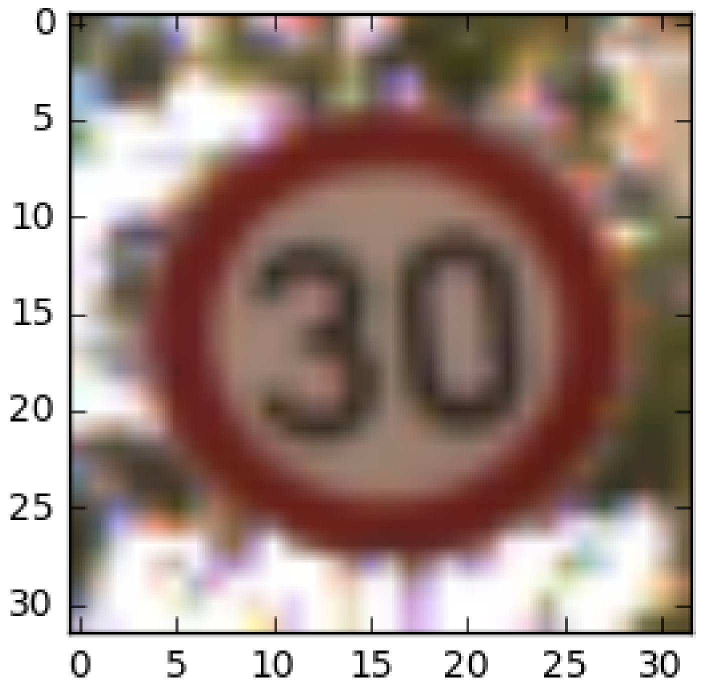
In [7]:

```
### Preprocess the data here. Preprocessing steps could include normalization
, converting to grayscale, etc.
### Feel free to use as many code cells as needed.

from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split


processed_train = np.array([prep_img(X_train[i]) for i in range(len(X_train))
], dtype = np.float32)
processed_test = np.array([prep_img(X_test[i]) for i in range(len(X_test))],
dtype = np.float32)
X_test = processed_test

print(len(processed_train))
print(len(X_train))
print(len(train['labels']))

image_post = processed_train[index]


##Split and Shuffle the images

X_train, X_validation, y_train, y_validation = train_test_split(processed_tra
in,train['labels'])
```

```
39209
39209
39209
```

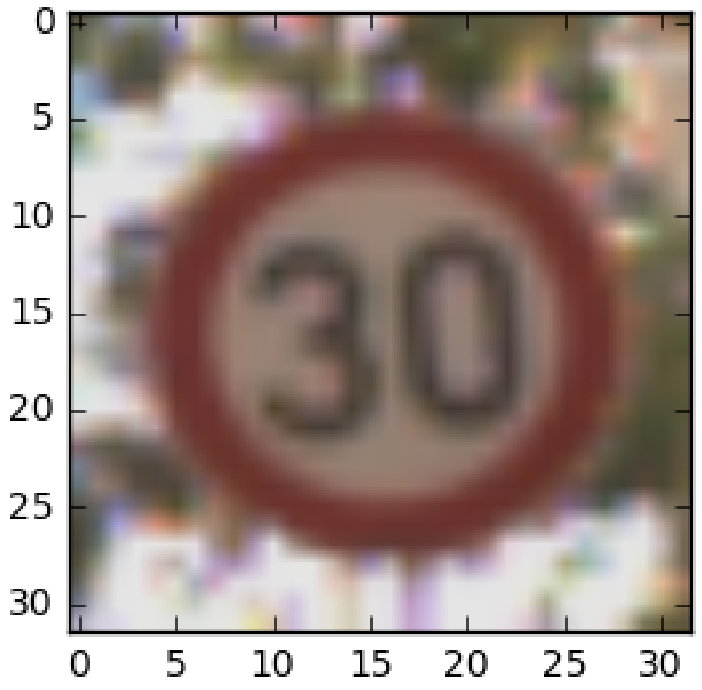
In [8]:
```
#index = random.randint(0, len(X_train))
plt.figure(figsize=(3,3))
plt.imshow(image_pre, cmap="gray")
```

Out[8]:
```
<matplotlib.image.AxesImage at 0x125d65198>
```



In [9]:
```
post_proc = prep_img(image_pre)
plt.figure(figsize=(3,3))
plt.imshow(post_proc)
```

Out[9]:
```
<matplotlib.image.AxesImage at 0x125e10518>
```



Model Architecture

WRITEUP

3. Describe, and identify where in your code, what your final model architecture looks like including model type, layers, layer sizes, connectivity, etc.) Consider including a diagram and/or table describing the final model.

The code for my final model is located in the 2 code cells that follow. I used architecture heavily based on the Le Net architecture, i.e. a convo net which seems to be very well suited for image classification tasks such as this. I've added two dropouts, to reduce the number of parameters and prevent over-fitting, and therefore make the model more robust.

My final model consisted of the following layers:

Layer	Description
Input	32x32x3 RGB image
Convolution 5x5	1x1 stride, valid padding, outputs 28x28x6
RELU	Activation function
Max pooling	2x2 stride, outputs 14x14x6
Convolution 5x5	1x1 stride, valid padding, outputs 10x10x16
RELU	Activation function
Max pooling	2x2 stride, outputs 5x5x16
Flatten	Flattens the layer to prep it for fully conn.
Fully connected	Used xW + b function, output 120
RELU	Activation function
1st dropout	
Fully connected	Input 120, out 84, xW + b function
RELU	Activation function
2nd dropout	
Output	Final fully connected layer. Outputs 43

WRITEUP

1. Describe how, and identify where in your code, you trained your model. The discussion can include the type of optimizer, the batch size, number of epochs and any hyperparameters such as learning rate.

The cell below contains my main hyperparameters. Through experimentation, I've settled at 15 Epochs, at which stage the improvement to model performace was marginal. I've also used a really low learning rate of 0.0005, as I've found out that while this makes the model really underperfom in the first few epochs, it leads to a more robust result in the end. I've used a default (Le Net) batch size of 128, as I didn't see any significant performance gains or losses when experimenting with the parameter. I've also used the keep_prob parameter value of 0.6 for dropout function, which I have found performed better than the 50% probability.

The actual training of the model occurs 6 code cells down. I've used adam optimizer, which performed better than a traditional gradient descent one.

In [10]:
```
##Set up tensorflow and main hyperparameners

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


EPOCHS = 15
BATCH_SIZE = 128
learning_rate = 0.0005
```

In [11]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.

def NetLaNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variables for
the weights and biases for each layer
    mu = 0
    sigma = 0.1

    # Layer 1: Convolutional. Input = 32x32x3. 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

    # Activation functions. Went for ReLu
    conv1 = tf.nn.relu(conv1)

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

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

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

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

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

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

    # Dropout - to prevent overfitting.
    fc1 = tf.nn.dropout(fc1, keep_prob)

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

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

    # Dropout - to prevent overfitting.
    fc2 = tf.nn.dropout(fc2, keep_prob)

    # Layer 5: Fully Connected. Input = 84. Output = 43.
    fc3_W  = tf.Variable(tf.truncated_normal(shape=(84, 43), mean = mu, stdde
v = 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 test set but low accuracy on the validation set implies overfitting.

In [13]:
```
##ONE-HOT-ENCODE and PREP VARIABLES


from datetime import datetime

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

In [14]:
```
##TRAIN PIPELINE

logits = NetLaNet(x)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, one_hot_y)
loss_operation = tf.reduce_mean(cross_entropy)
optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate)
training_operation = optimizer.minimize(loss_operation)
```

In [15]:
```
##EVALUATE PIPELINE

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:of
fset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x, y: bat
ch_y, keep_prob : 1.0})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [16]:
```
##TRAINING THE MODEL

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    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, k
eep_prob : 0.6})

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

    filename = './net/netlanet_03'
    saver.save(sess, filename)
    print("Model saved")
```

```
Training...

EPOCH 1 ...
Validation Accuracy = 0.399

EPOCH 2 ...
Validation Accuracy = 0.562

EPOCH 3 ...
Validation Accuracy = 0.726

EPOCH 4 ...
Validation Accuracy = 0.810

EPOCH 5 ...
Validation Accuracy = 0.856

EPOCH 6 ...
Validation Accuracy = 0.884

EPOCH 7 ...
Validation Accuracy = 0.898

EPOCH 8 ...
Validation Accuracy = 0.924

EPOCH 9 ...
Validation Accuracy = 0.932

EPOCH 10 ...
Validation Accuracy = 0.938

EPOCH 11 ...
```

```
Validation Accuracy = 0.942

EPOCH 12 ...
Validation Accuracy = 0.950

EPOCH 13 ...
Validation Accuracy = 0.955

EPOCH 14 ...
Validation Accuracy = 0.960

EPOCH 15 ...
Validation Accuracy = 0.962

Model saved
```

In [17]:
```
#Launch the model on the test data
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    test_accuracy = sess.run(accuracy_operation, feed_dict={x: X_test, y: y_t
est, keep_prob : 1.0})

print('Test Accuracy: {}'.format(test_accuracy))
```

```
Test Accuracy: 0.9008709192276001
```

WRITEUP

1. Describe the approach taken for finding a solution. Include in the discussion the results on the training, validation and test sets and where in the code these were calculated.

> Your approach may have been an iterative process, in which case, outline the steps you took to get to the final solution and why you chose those steps. Perhaps your solution involved an already well known implementation or architecture. In this case, discuss why you think the architecture is suitable for the current problem.

The code for calculating the accuracy of the model is located in the two code cells above.

My final model results were:

- validation set accuracy of 96.2% (after 15 Epochs)
- test set accuracy of 90.1%

I have started with the Le Net architecture, which is a simple but robust architecture for such image clasification problems. From there, I've taken an iterative approach, tweaking in turn one (or sometimes several) of the following parameters:

- Hyperparameters. I've experimented with learning rates from 0.1 (way too high) to 0.0001 (too slow and inefficient). I've experimented with expanding the number of epochs up to 30, and experimented with the keep_prob value of 0.3 to 0.75. I've kept or discarded changes based on the performace with the validation set + 1 or 2 randomly chosen pictures from the internet (different ones were used every time)
- Image pre-processing pipeline - as described in the section above - I've experimented with several techniques, but in the end kept just the simple normalization, which seems to work and I liked the simplicity.
- Network architecture, in particular different activation functions, different padding and stride parameters, but also addition of new layers (e.g., dropout)

The initial architecture was performing well off the shelf, reaching ~94% accuracy on the validation set. The reason this architecture performs well is that it's a convo net, which classifies features of an image without regarding their spatial position on the image - making it powerful to extract just the features we're interested in, regardless of where exactly in the picture they can be found. However, the model underperformed when I tested it on new random pictures from the internet - indicating the model was over-fitting. I've therefore added two dropout functions to the final architecture which made the model more robust and prevented the overfitting. The final model reached a test set validation of ~90% whcih I was happy with.

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.

WRITEUP

1. Choose five German traffic signs found on the web and provide them in the report. For each image, discuss what quality or qualities might be difficult to classify.
2. The first image ("no vehicles") should be easy to clasify. It's a class with many examples and a fairly distinctive sign. I wanted to give the model something fairly easy to predict as the first task, and see how confident it would be.
3. The second image ("right only") could be difficult due to it's similarity with the "left only" sign, as well as the image quality degraded by dirt and watermark. I also wanted to see how the model would do on a blue sign, as I explicitly kept the model working with 3-channel images instead of greyscale ones, to preserve color information which could be

useful, such as in this case.

4. The third image ("no vehicles over 3.5 tons") was chosen to see how well the model could differentiate between this sign and the speed limit signs (or other signs where the general feature is a red circle and something depicted on the white center of the sign). This is one of the less popular image classes, so I was wondering if the model could distinguish it from the more frequent speed sign classes (Spoiler alert: it couldn't).
5. Fourth image ("Yield") - a frequent and distinctive sign, I expected no trouble with classification.
6. Fifth image ("Slippery road"). This will be a difficult one to classify, as it's a sign class with very low frequency, which will compete with other triangular warning signs. Also, this image does not have square aspect ratio, and the actual sign only covers small portion of the canvas (relative to what the model saw in the training set.) I wanted to see how the model could deal with such difficult picture, and if the correct answer at least made it into the top 5 probabilities.

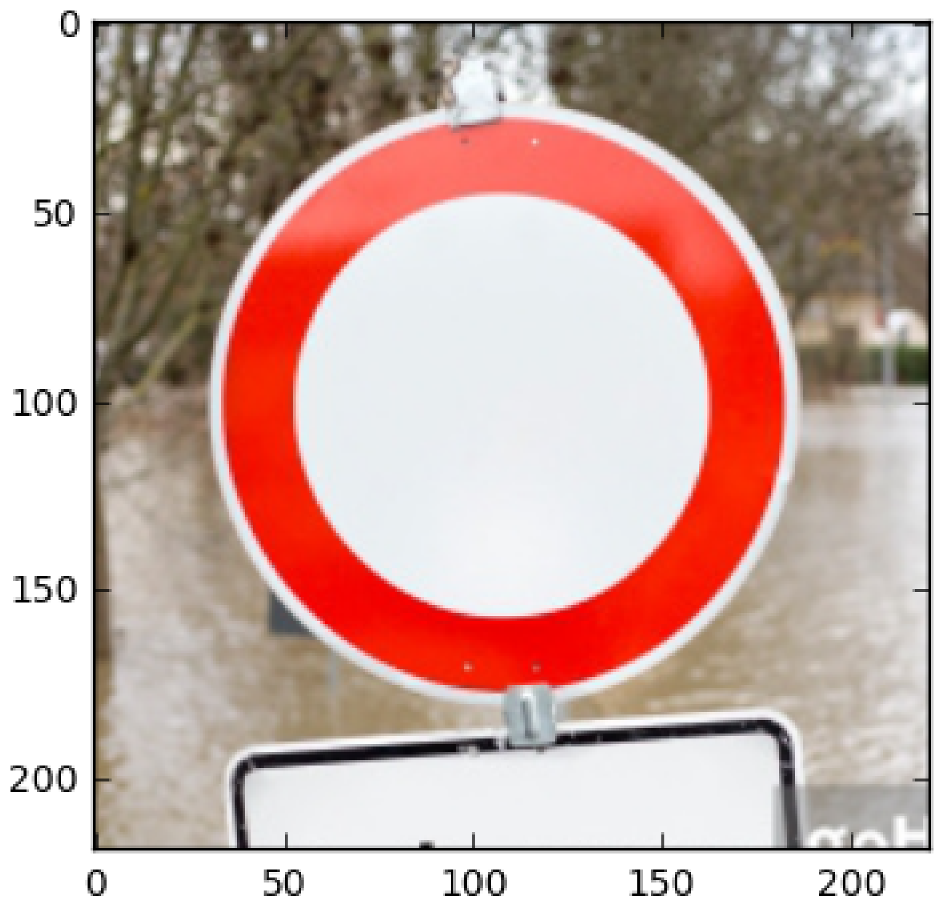
Load and Output the Images

In [18]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
```

In [19]:
```
##Helper to change the loaded images to RGB scheme

import cv2

def rgb(img):
    b,g,r = cv2.split(img)
    img2 = cv2.merge([r,g,b])
    return img2
```

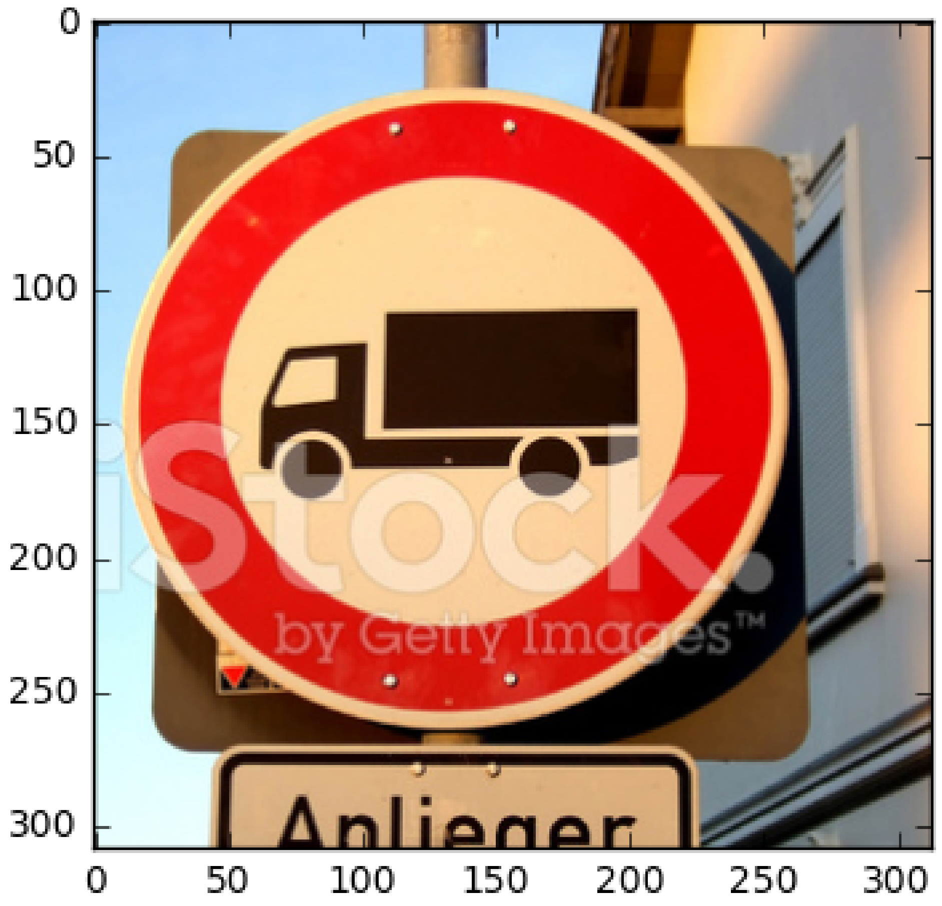
In [20]:
```
image1 = rgb(cv2.imread('./unseen/01-noentry.jpg'))
image2 = rgb(cv2.imread('./unseen/02-way.jpg'))
image3 = rgb(cv2.imread('./unseen/03-35tons.jpg'))
image4 = rgb(cv2.imread('./unseen/04-giveway.jpg'))
image5 = rgb(cv2.imread('./unseen/05-slipperyroad.jpg'))
```

In [21]:
```
plt.imshow(image1)
plt.show()
```



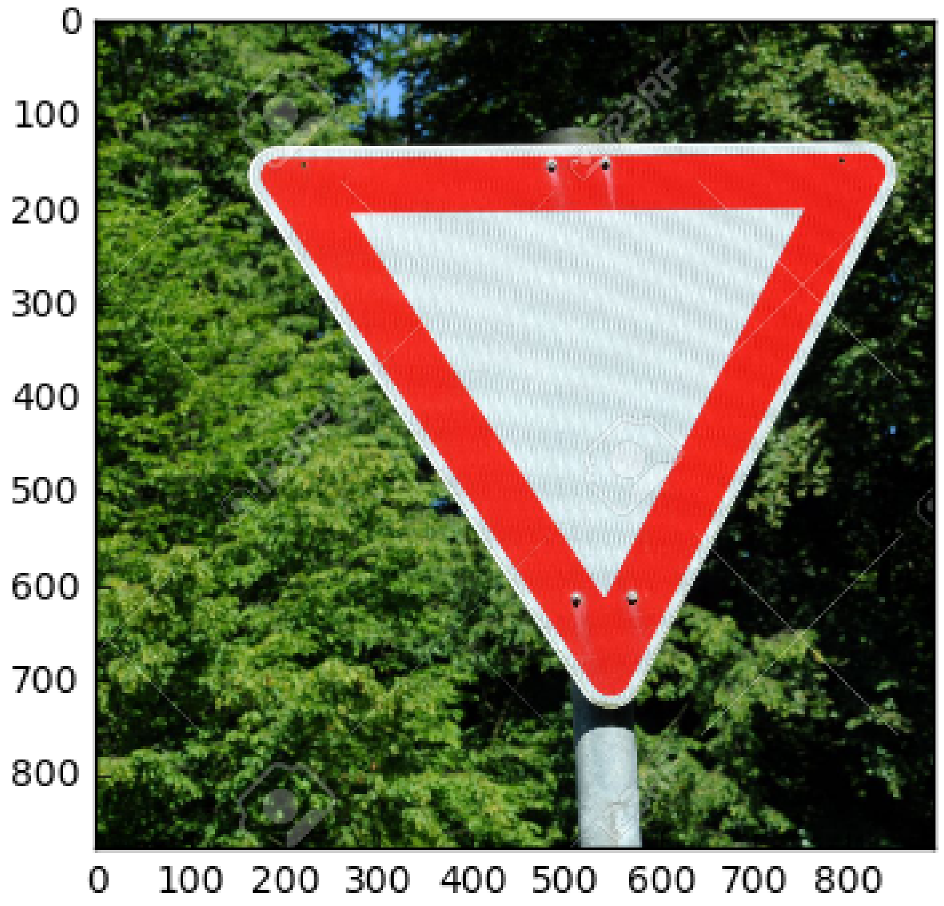
In [22]:
```
plt.imshow(image2)
plt.show()
```

In [23]:
```
plt.imshow(image3)
plt.show()
```



Predict the Sign Type for Each Image

In [24]:
```
plt.imshow(image4)
plt.show()
```



In [25]:
```
plt.imshow(image5)
plt.show()
```

WRITEUP

1. Discuss the model's predictions on these new traffic signs and compare the results to predicting on the test set. Identify where in your code predictions were made. At a minimum, discuss what the predictions were, the accuracy on these new predictions, and compare the accuracy to the accuracy on the test set (OPTIONAL: Discuss the results in more detail as described in the "Stand Out Suggestions" part of the rubric).

The cells below contain the predictions as well as the softmax probabilities, along with the images displayed in resized, post-processed version, i.e., same as seen by the convo net.

The results of the test are as follows:

Sign	Prediction	Top 5 predictions
No vehicles	No vehicles	No vehicles(99.9%),No passing, Speed limit 120km/h, Speed limit 50 km/h, Yield
Keep right	Keep right	Keep right (99.9%), Roundabout mandatory, Turn left ahead, Dangerous curve to the right, Go straight or right
No vehicles over 3.5 tons	Speed lim. 50km/h	Speed limit 50 km/h (63.8%), Speed limit 30 km/h (17%), Speed limit 100 km/h (9%), 80 km/h (4%), 120 km/h (2%)
Yield	Yield	Yield (99.9%), No passing, No passing over 3.5t, Priority road,Speed lim. 60km/h
Slippery road	Bumpy road	Bumpy road (92.6%), Bicycles crossing (3.9%) Road work, Traffic signals,

```
|                  |                  |Wild animals crossing
---------------------------------------------------------------------
```

The model was able to correctly guess 3 of the 5 traffic signs, which gives an accuracy of 60%. I think this is a solid result, given that my initial pipeline was often returning 0 - 40% accuracy on similar tests.

COMPARISON TO TEST RESULTS

The accuracy on the new captured images was 60%, compared to accuracy of ~90% on the test set. This suggests that the model is overfitting, and has not really generalized the knowledge of traffic signs to the extent the testing and validation sets would suggest. To remedy this, probably additional training data would be helpful - esp. ones generated through augmentation by jittering and transforming the images, and scaling them up and down.

3. Describe how certain the model is when predicting on each of the five new images by looking at the softmax probabilities for each prediction and identify where in your code softmax probabilities were outputted. Provide the top 5 softmax probabilities for each image along with the sign type of each probability. (OPTIONAL: as described in the "Stand Out Suggestions" part of the rubric, visualizations can also be provided such as bar charts)

The softmax probabilities can be found directly under the picture in the code cells below. I've also highlighted the probabilities in the table above.

- For the first image, the model is very certain it has the correct sign. Other alternatives have negligible probability, and are predicted based on the shape of the sign.
- For the second image, the model is also very certain (99.9%). What I found surprising is that the mirror image ("keep left") is not even in the top 5 softmax probabilities, indicating the direction of the arrow has significant influence on the prediction (i.e., the model is not

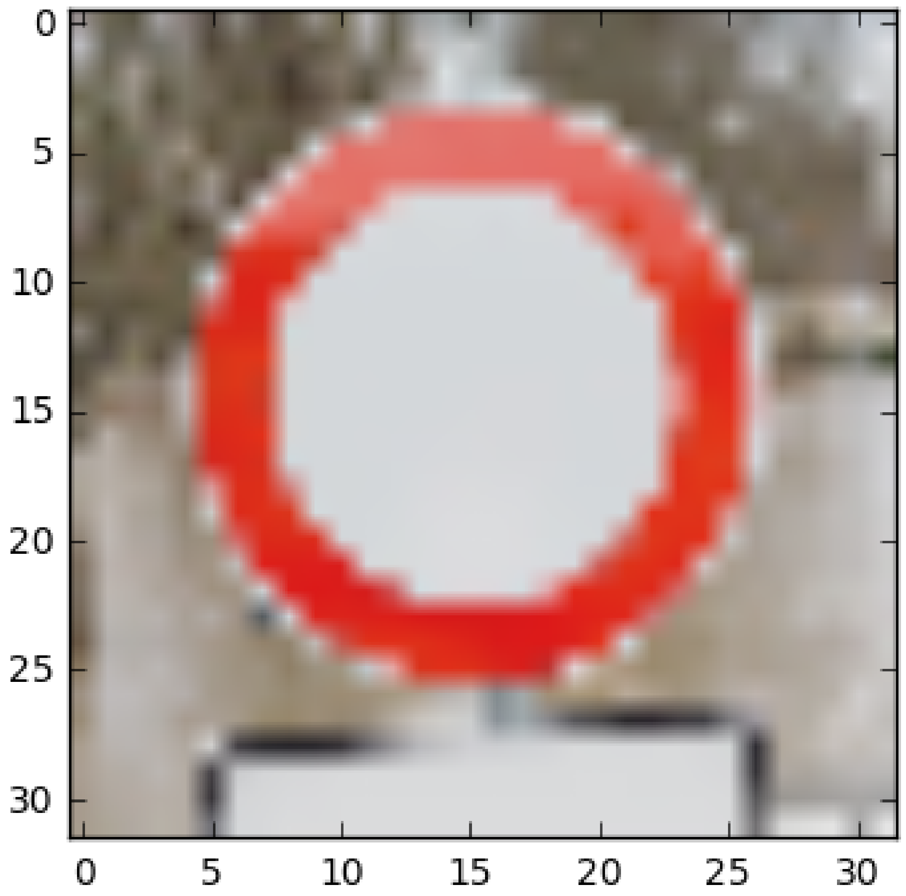
as easily confused as I thought it might be.)

- For the third image, the model gets it wrong - and we see a much more even distribution of probabilities: first guess is only ~64% and second (also wrong) about 17%. It's disappointing to see that the correct answer is not in top 5 guesses.
- For the fourth image, same as cases 1 and 2 - the model is very very certain. Other guesses again reflect the sign's shape - in this case a triangle.
- Fifth image, slippery road: The prediction is for bumpy road, which is sign that is visually very similar. I'm somewhat surprised by the relatively high probability (~93%), especially because I consider the next choices (e.g., bicycle crossing) also to be quite visually similar. This again could be helped by introducing more training data, ideally with random angle rotation and jitter - as the model is not doing great at recognizing less frequent classes.

In summary, it is clear that the model could benefit from more training data for some of the less frequent / common classes. I would do this by implementing random angle rotation + some transform or offset function. I would generate new images to fill each class to a minimum number of samples. This would potentially make the model more robust by introducing more examples for the less frequent classes, but would also keep some level of differentiation between the most frequent classes and the least frequent ones, thereby keeping this useful information for the model to extract. The above will be my first improvement I'll make to the model when I have more capacity.

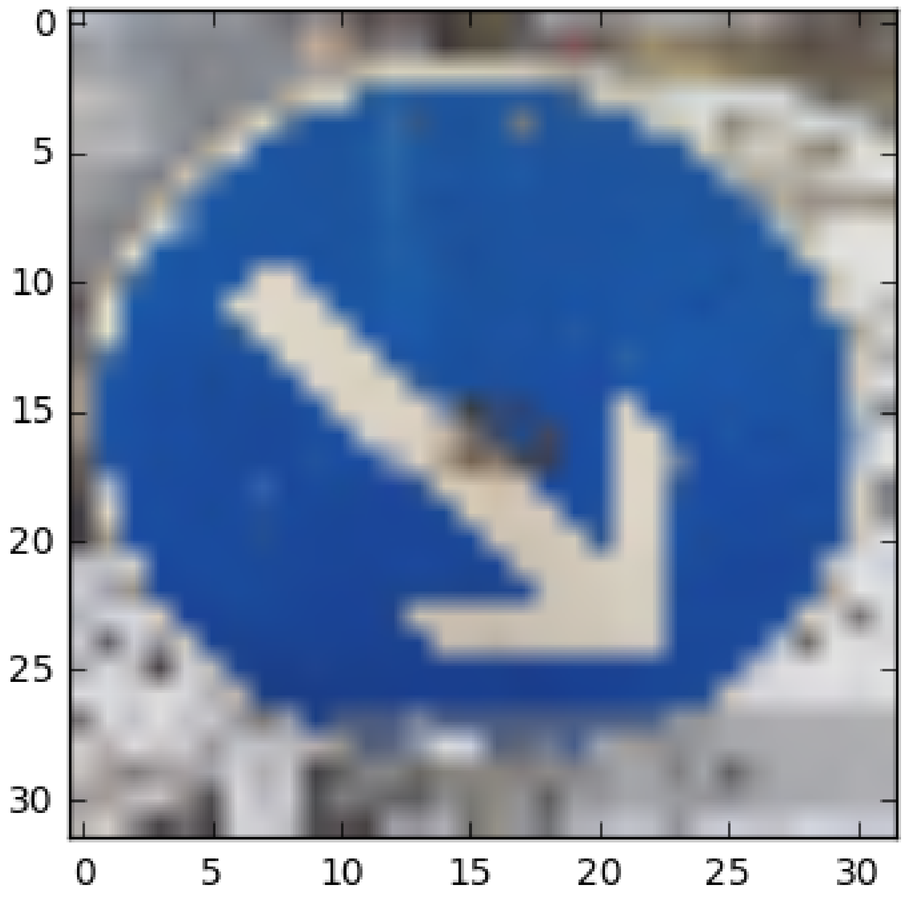
In [26]:
```
### Run the predictions here and use the model to output the prediction for e
ach image.
### Make sure to pre-process the images with the same pre-processing pipeline
used earlier.
### Feel free to use as many code cells as needed.


image = cv2.resize(image1,(32,32))
image = prep_img(image)

plt.imshow(image)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    new_img = np.expand_dims(image,axis=0)

    predict = tf.argmax(logits, 1)
    probabilities = logits


    print("Prediction: ", predict.eval(feed_dict = {x : new_img, keep_prob :
1.0 }))
    print("Top 5: ", sess.run(tf.nn.top_k(tf.nn.softmax(logits.eval(feed_dict
= {x : new_img, keep_prob : 1.0 })),5)))
```

```
Prediction:  [15]
Top 5:  TopKV2(values=array([[  9.99998331e-01,   1.18835203e-06,   4.1554386
6e-07,
          6.32811918e-08,   1.77183495e-08]], dtype=float32), indices=array([
[15,  9,  8,  2, 13]], dtype=int32))
```


0
5
10
15
20
25
30
0
5
10
15
20
25
30

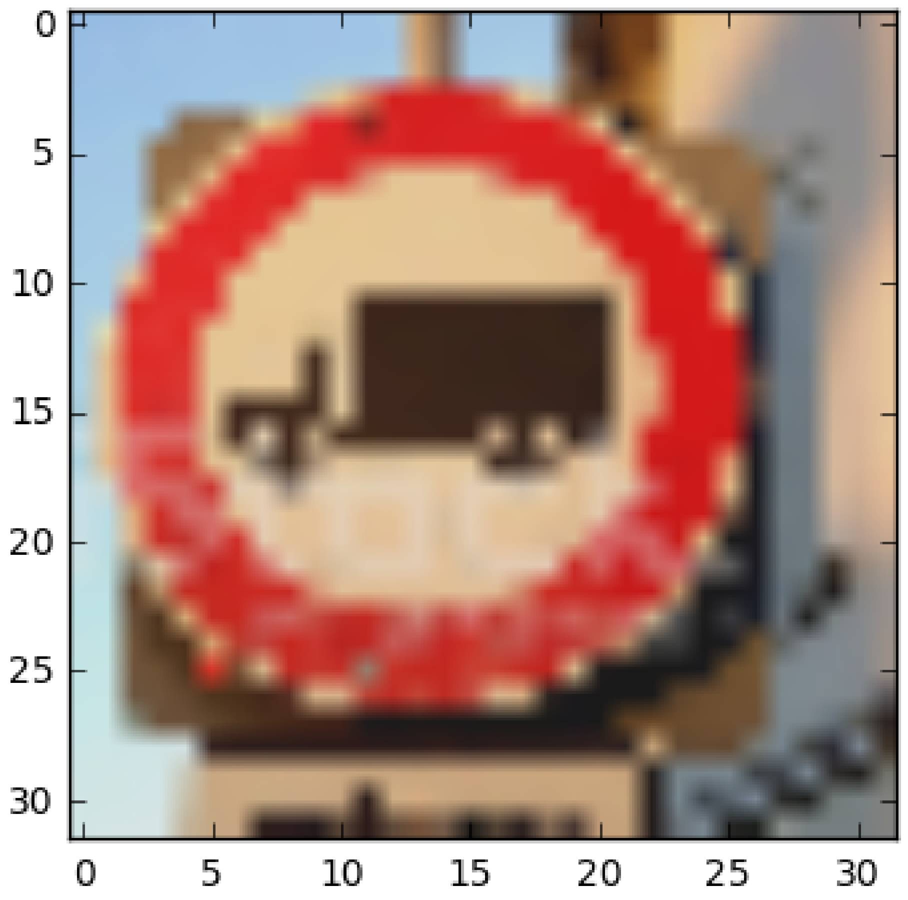
In [27]:
```
### Run the predictions here and use the model to output the prediction for e
ach image.
### Make sure to pre-process the images with the same pre-processing pipeline
used earlier.
### Feel free to use as many code cells as needed.


image = cv2.resize(image2,(32,32))
image = prep_img(image)

plt.imshow(image)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    new_img = np.expand_dims(image,axis=0)

    predict = tf.argmax(logits, 1)
    probabilities = logits


    print("Prediction: ", predict.eval(feed_dict = {x : new_img, keep_prob :
1.0 }))
    print("Top 5: ", sess.run(tf.nn.top_k(tf.nn.softmax(logits.eval(feed_dict
= {x : new_img, keep_prob : 1.0 })),5)))
```

```
Prediction:  [38]
Top 5:  TopKV2(values=array([[  9.99996781e-01,   3.06336483e-06,   8.0777361
7e-08,
          2.10219925e-10,   1.86920163e-12]], dtype=float32), indices=array([
[38, 40, 34, 20, 36]], dtype=int32))
```


0
5
10
15
20
25
30
0
5
10
15
20
25
30

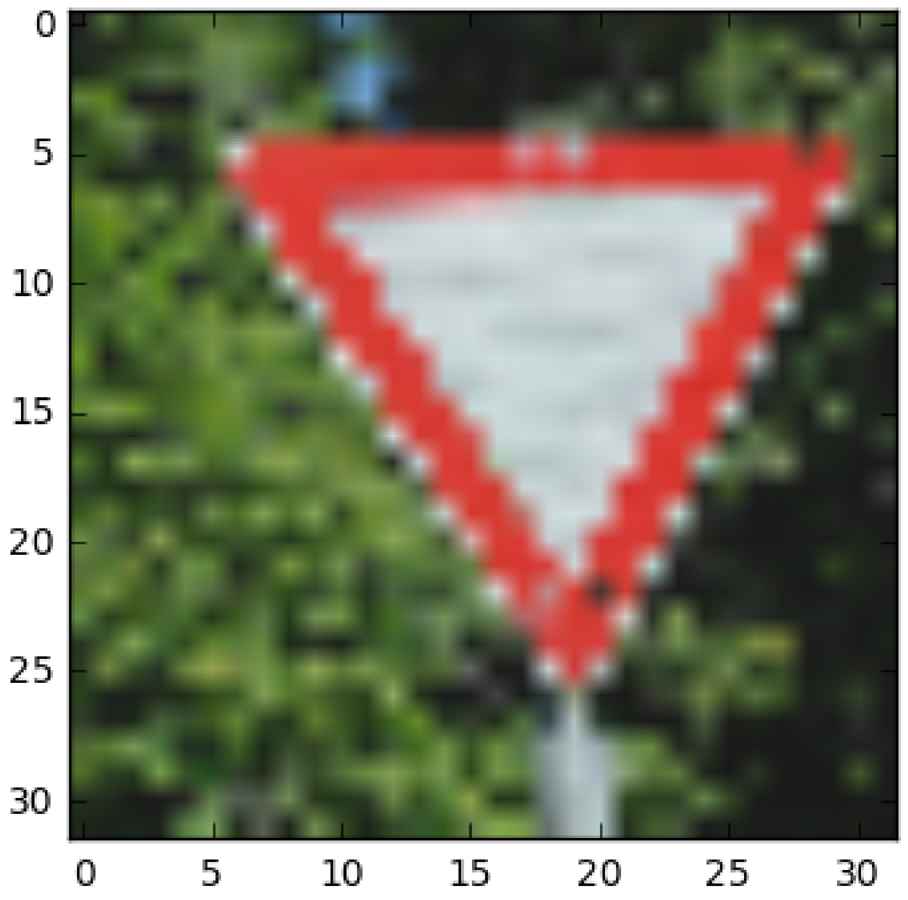
In [28]:
```
### Run the predictions here and use the model to output the prediction for e
ach image.
### Make sure to pre-process the images with the same pre-processing pipeline
used earlier.
### Feel free to use as many code cells as needed.


image = cv2.resize(image3,(32,32))
image = prep_img(image)

plt.imshow(image)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    new_img = np.expand_dims(image,axis=0)

    predict = tf.argmax(logits, 1)
    probabilities = logits


    print("Prediction: ", predict.eval(feed_dict = {x : new_img, keep_prob :
1.0 }))
    print("Top 5: ", sess.run(tf.nn.top_k(tf.nn.softmax(logits.eval(feed_dict
= {x : new_img, keep_prob : 1.0 })),5)))
```

```
Prediction:  [2]
Top 5:  TopKV2(values=array([[ 0.63844633,  0.1764199 ,  0.09371448,  0.04396
971,  0.01673588]], dtype=float32), indices=array([[2, 1, 7, 5, 8]], dtype=in
t32))
```



In [29]:
```
### Run the predictions here and use the model to output the prediction for e
ach 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.



image = cv2.resize(image4,(32,32))
image = prep_img(image)

plt.imshow(image)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    new_img = np.expand_dims(image,axis=0)

    predict = tf.argmax(logits, 1)
    probabilities = logits


    print("Prediction: ", predict.eval(feed_dict = {x : new_img, keep_prob :
1.0 }))
    print("Top 5: ", sess.run(tf.nn.top_k(tf.nn.softmax(logits.eval(feed_dict
= {x : new_img, keep_prob : 1.0 })),5)))
```

```
Prediction:  [13]
Top 5:  TopKV2(values=array([[  9.99994516e-01,   4.95367203e-06,   5.0916025
9e-07,
          6.97251146e-10,   2.05625267e-10]], dtype=float32), indices=array([
[13,  9, 10, 12,  3]], dtype=int32))
```


0
5
10
15
20
25
30
0
5
10
15
20
25
30

In [30]:
```
### Run the predictions here and use the model to output the prediction for e
ach 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.



image = cv2.resize(image5,(32,32))
image = prep_img(image)

plt.imshow(image)

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('./net'))

    new_img = np.expand_dims(image,axis=0)

    predict = tf.argmax(logits, 1)
    probabilities = logits


    print("Prediction: ", predict.eval(feed_dict = {x : new_img, keep_prob :
1.0 }))
    print("Top 5: ", sess.run(tf.nn.top_k(tf.nn.softmax(logits.eval(feed_dict
= {x : new_img, keep_prob : 1.0 })),5)))
```

```
Prediction:  [22]
Top 5:  TopKV2(values=array([[ 0.92606831,  0.03940045,  0.01805967,  0.00679
721,  0.00274941]], dtype=float32), indices=array([[22, 29, 25, 26, 31]], dty
pe=int32))
```

0
5
10
15
20
25
30
0
5
10
15
20
25
30