CHUCK-P / Traffic_Sign_Classifier

Unwatch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Settings | Insights

Branch: master | Traffic_Sign_Classifier / Traffic_Sign_Classifier.ipynb | Find file | Copy path

CHUCK-P final Traffic Sign aa66d91 9 hours ago

8 contributors

Executable File 1275 lines (1274 sloc) 248 KB | Raw | Blame | History

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 [12]:
```
# Load pickled data
import pickle

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

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

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

X_train, y_train = train['features'], train['labels']
X_valid, y_valid = valid['features'], valid['labels']
X_test, y_test = test['features'], test['labels']
```

Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

- `'features'` is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
- `'labels'` is a 1D array containing the label/class id of the traffic sign. The file `signnames.csv` contains id -> name mappings for each id.
- `'sizes'` is a list containing tuples, (width, height) representing the original width and height the image.
- `'coords'` is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below. Use python, numpy and/or pandas methods to calculate the data summary rather than hard coding the

results. For example, the pandas shape method (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shape.html) might be useful for calculating some of the summary results.

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

In [13]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the results
import pandas as pd
import numpy as np

# TODO: Number of training examples
n_train = len(X_train)
n_y_train = len(y_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 = len(np.unique(y_train))

print("Number of training examples =", n_train)
print("Number of y training labels =", n_y_train)
print("Number of validation 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 y training labels = 34799
Number of validation examples = 4410
Number of testing examples = 12630
Image data shape = (32, 32, 3)
Number of classes = 43
```

Include an exploratory visualization of the dataset

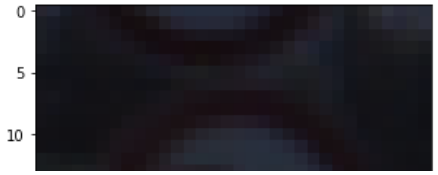
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

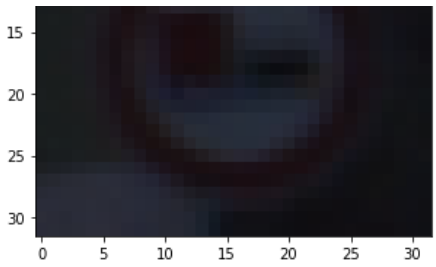
The Matplotlib (http://matplotlib.org/) examples (http://matplotlib.org/examples/index.html) and gallery (http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. 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 [14]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import random
import matplotlib.pyplot as plt
# Visualizations will be shown in the notebook.
%matplotlib inline

index = random.randint(0, len(X_train))
image = X_train[index].squeeze()

plt.figure(figsize=(5,5))
plt.imshow(image)
print(y_train[index])
```

```
10
```



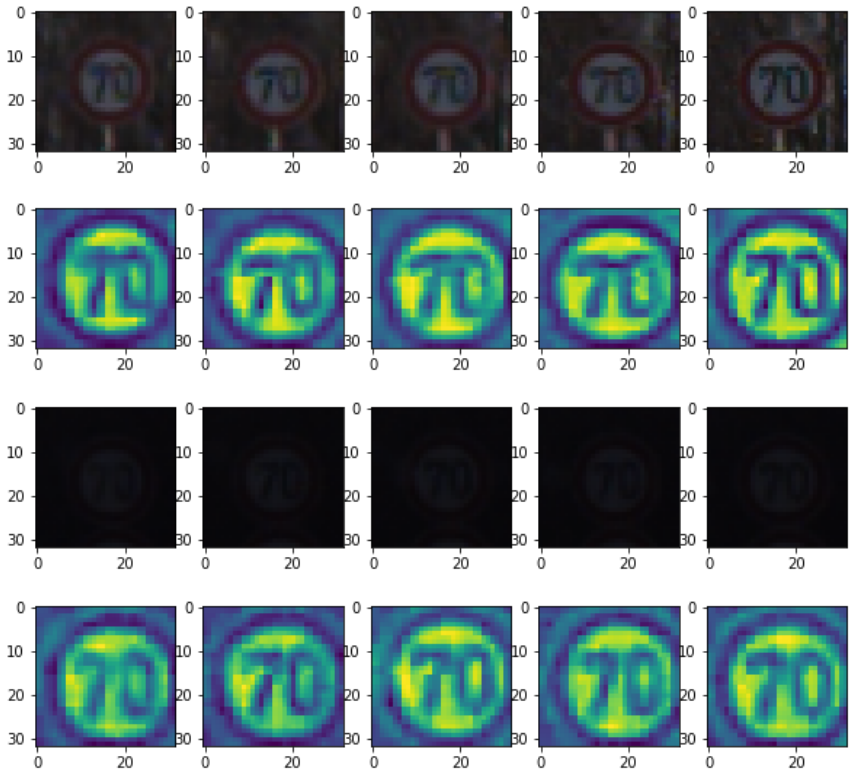


In [15]:
```
fig,ax = plt.subplots()
#histogram of the data
n, bins, patches = ax.hist(y_train,n_classes) #normed=1
ax.set_xlabel('Training Dataset Class')
ax.set_ylabel('Density')
ax.set_title('Histogram of the training data unique classes')
#
fig.tight_layout()
plt.show()

fig.savefig('./examples/plot1.png')
```

In [16]:
```
# FUNCTIONS

# import packages
from scipy import ndimage
import cv2

# from numpy import newaxis HOLD

# Normalize the training data and the validation data to be between -1 and 1 (-1 and .992) from single cha
nnel data 0 to 255
def normalize(image):
    image = (image - 127.5) / 127.5
    return image

def resizeImage(image):
    return cv2.resize(image,(32,32),interpolation = cv2.INTER_AREA)

def cropImage(input):
    roi = input[7:25,7:25]
    return resizeImage(roi)

def sharpen(image):
    img = cv2.GaussianBlur(image,(9,9), 10);
    img1 =  cv2.addWeighted(image, 2, img, -1, 0);
    return img1

def Transform(image,rot):
    return resizeImage(ndimage.rotate(image, rot))

# PREPROCESS PIPELINE
def pipeline(input):
    # 1. Convert input RGB image to grayscale
    # 2. Normalize image range -1.0 to 1.0
    # 3. Crop image to remove unnecessary pixels
    # 4. Sharpen the cropped image
    gray = cv2.cvtColor(input,cv2.COLOR_BGR2GRAY)
    norm = normalize(gray)
    crop = cropImage(norm)
```

```
    sharp = sharpen(crop)
    return sharp
```

In [17]:
```
# Display random 5 images

index = 4 # Choose a label to draw

disp = [X_train[i] for i in range(len(y_train)) if (y_train[i]==index)]
valid = [X_valid[i] for i in range(len(y_valid)) if (y_valid[i]==index)]

fig = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(disp[j])

fig1 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(disp[j]))

fig2 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(valid[j])

fig3 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(valid[j]))
```



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.

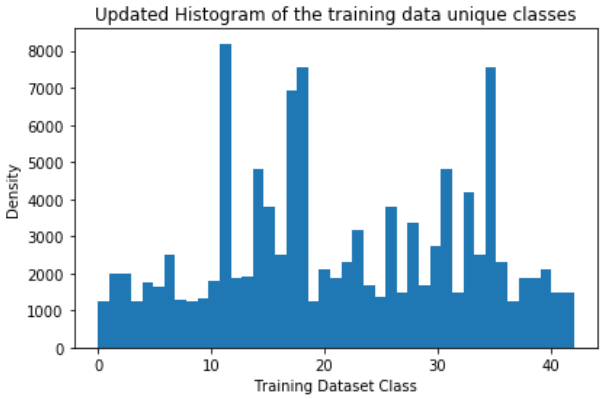
In [18]:
```
### Preprocess the data here. It is required to normalize the data. Other preprocessing steps could includ
e
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.


# Preprocess all data
# set up gray scaled placeholder images
X_train_gray = np.zeros((len(X_train),32,32))
X_valid_gray = np.zeros((len(X_valid),32,32))
X_test_gray = np.zeros((len(X_test),32,32))

# populate the placeholders
for i in range(len(X_train)):
    X_train_gray[i] = pipeline(X_train[i])

for i in range(len(X_valid)):
    X_valid_gray[i] = pipeline(X_valid[i])

for i in range(len(X_test)):
    X_test_gray[i] = pipeline(X_test[i])

# Normalize the training data and the validation data to be between -1 and 1 (-1 and .992) from single cha
nnel data 0 to 255
#for i in [X_train, y_train, X_valid, y_valid]:
#    for pixel in i:
        # was pixel = (pixel - 128) / 128
        #pixel = (pixel - 128.) / 128.
#        pixel = (pixel - 127.5) / 127.5

#plt.imshow(image)
```

In [19]:
```
# Add more data for low sample lables
# For each data label, add 6 more images per image - *JUST FOR TRAINING DATA*
index_count = np.bincount(y_train)
indexes = [i for i in range(len(index_count)) if index_count[i] < 1200]
print("Labels that require more data:", indexes)
angles = [-20,-15,-10,10,15,20]

# Collect all class labels that require data generation
X_data, y_data = zip(*[(X_train_gray[i], y_train[i]) for i in range(0,len(y_train)) if y_train[i] in index
es])

# Compute rotation images for X_data
X_rotate, y_rotate = zip(*[(Transform(X_data[i],rotation), y_data[i]) for rotation in angles for i in rang
e(0,len(y_data))])

# Append the newly generated data to the original data
X_train_new = np.append(X_train_gray,X_rotate,axis=0)
y_train = np.append(y_train,y_rotate,axis=0)

# Print stuff
print("Additional data samples generated: ", np.shape(X_rotate[0]))
print("New size - X_train: ", np.shape(X_train_new))
print("New size - y_train: ", np.shape(y_train))
```

```
Labels that require more data: [0, 6, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 30,
 31, 32, 33, 34, 35, 36, 37, 39, 40, 41, 42]
Additional data samples generated:  (32, 32)
New size - X_train:  (115433, 32, 32)
New size - y_train:  (115433,)
```

In [20]:
```
# SHUFFLE AND RESHAPE THE DATA

# import packages
from sklearn.utils import shuffle
from numpy import newaxis

# First conv layer in tensorflow requires data to be in (none,32,32,1) structure - ADDED
X_train_new = X_train_new[..., newaxis]
X_valid = X_valid_gray[..., newaxis]
X_test = X_test_gray[..., newaxis]


# Convert to Grayscale to make it easier to use the MNIST classifier
#gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Shuffle the training data
X_train_new, y_train = shuffle(X_train_new, y_train)
# Shuffle the validation data
X_valid, y_valid = shuffle(X_valid, y_valid)
# Shuffle the test data
X_test, y_test = shuffle(X_test, y_test)
```

In [21]:
```
# PLOT UPDATED HISTOGRAM
fig,ax = plt.subplots()
#histogram of the data
n, bins, patches = ax.hist(y_train,n_classes) #,normed=1
ax.set_xlabel('Training Dataset Class')
ax.set_ylabel('Density')
ax.set_title('Updated Histogram of the training data unique classes')
#
fig.tight_layout()
plt.show()

fig.savefig('./examples/plot2.png')
```



Model Architecture

In [22]:
```
### Define your architecture here.
### Feel free to use as many code cells as needed.

# experiment with number of layers, number of filters, dropout

# Model Setup
import tensorflow as tf
from tensorflow.contrib.layers import flatten



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

```
    # Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6.  Changed from 3 input channels to 1
    conv1_W = tf.Variable(tf.truncated_normal(shape=(5, 5, 1, 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.
    conv1 = tf.nn.relu(conv1)

    # Dropout ADDED
    conv1 = tf.nn.dropout(conv1, keep_prob)

    # Pooling. Input = 28x28x6. Output = 14x14x6.
    conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='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='VALID') + conv2_b

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

    # Dropout ADDED
    conv2 = tf.nn.dropout(conv2, keep_prob)

    # Pooling. Input = 10x10x16. Output = 5x5x16.
    conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='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, stddev = sigma))
    fc1_b = tf.Variable(tf.zeros(120))
    fc1   = tf.matmul(fc0, fc1_W) + fc1_b

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

    #  ADDED 1st dropout per reference model
    #drop1 = 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, stddev = sigma))
    fc2_b  = tf.Variable(tf.zeros(84))
    # subbed drop1 for fc1
    fc2    = tf.matmul(fc1, fc2_W) + fc2_b

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

    # ADDED 2nd dropout per reference model
    #drop2 = 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, stddev = sigma))
    fc3_b  = tf.Variable(tf.zeros(43))

    # subbed drop2 for fc2
    logits = tf.matmul(fc2, fc3_W) + fc3_b



    return logits
```

Train, Validate and Test the Model

A validation set can be used to assess how well the model is performing. A low accuracy on the training and validation sets imply underfitting. A high accuracy on the training set but low accuracy on the validation set implies overfitting.

In [23]:
```
### 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.
```

```
from sklearn.utils import shuffle

# ADDED
keep_prob = tf.placeholder(tf.float32)

x = tf.placeholder(tf.float32, (None, 32, 32, 1)) # Changed to gray from RGB (3)
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 43)

# PARAMETERS
EPOCHS = 20 # EPOCHS default 10
BATCH_SIZE = 128 # 128 default

rate = 0.001 # LEARNING RATE  default .001

#keep_prob = .7 # 0.9 default USING "feed_dict" FOR EACH evaluate
```

In [24]:
```
# TRAINING PIPELINE



logits = LeNet(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)
training_operation = optimizer.minimize(loss_operation)
```

In [25]:
```
# MODEL EVALUATION

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, sess):
    num_examples = X_data.shape[0]
    total_accuracy = 0
    total_loss = 0 # ADDED
    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_SIZE]
        # OBTAIN loss and accuracy in same TF session - need to pass keep_prob?
        loss, accuracy = sess.run([loss_operation, accuracy_operation], feed_dict={x: batch_x, y: batch_y,
 keep_prob: 1.0})
        total_accuracy += (accuracy * batch_x.shape[0])
        total_loss += (loss * batch_x.shape[0])
    return total_loss / num_examples, total_accuracy / num_examples

# Return prediction and top_k values    ADDED
def predict(X_data, y_data):
    predictions = sess.run(tf.argmax(logits, 1),feed_dict={x: X_data, y: y_data,keep_prob: 1.0})
    softmax = tf.nn.softmax(logits)
    top_k = sess.run(tf.nn.top_k(softmax,k=5),feed_dict={x: X_data, y: y_data, keep_prob: 1.0})   #
    return predictions, top_k
```

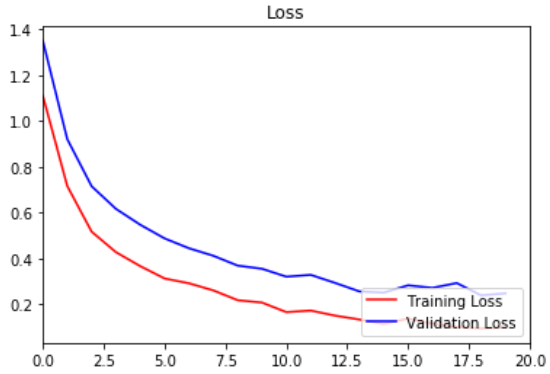
In [26]:
```
# TRAIN THE MODEL

# ADDED
train_loss_history = []
validation_loss_history = []
train_accuracy_history = []
validation_accuracy_history = []

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

    print("Training...")
    print()
    for i in range(EPOCHS):
        X_train_new, y_train = shuffle(X_train_new, y_train)    # CHANGED X_train to X_train_new - DNW
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train_new[offset:end], y_train[offset:end]  # CHANGED X_train to X_train_
new - DNW
            # need to ADD  keep_prob: 0.5
            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y, keep_prob: 0.5})

        # ADDED
```

```
        train_loss, train_accuracy = evaluate(X_train_new, y_train, sess) # CHANGED X_train to X_train_new
 - DNW
        train_loss_history.append(train_loss)
        train_accuracy_history.append(train_accuracy)
        validation_loss, validation_accuracy = evaluate(X_valid, y_valid, sess)
        validation_loss_history.append(validation_loss)
        validation_accuracy_history.append(validation_accuracy)

        print("EPOCH {} ...".format(i+1))
        print("Training Accuracy = {:.3f},".format(train_accuracy)) # ADDED
        print("Validation Accuracy = {:.3f}".format(validation_accuracy))
        print()

         # Evaluate on the test data
    #test_loss, test_acc = evaluate(X_test, y_test, sess)
    #print("Test loss = {:.3f}".format(test_loss))
    #print("Test accuracy = {:.3f}".format(test_acc))

    #loss_plot = plt.subplot(2,1,1)
    #loss_plot.set_title('Loss')
    #loss_plot.plot(training_loss_history, 'r', label='Training Loss')
    #loss_plot.plot(validation_loss_history, 'b', label='Validation Loss')
    #loss_plot.set_xlim([0, EPOCHS])
    #loss_plot.legend(loc=4)
    #acc_plot = plt.subplot(2,1,2)
    #acc_plot.set_title('Accuracy')
    #acc_plot.plot(training_accuracy_history,'r', label='Training Accuracy')
    #acc_plot.plot(validation_accuracy_history, 'b', label='Validation Accuracy')
    #acc_plot.set_ylim([0, 1.0])
    #acc_plot.set_xlim([0, EPOCHS])
    #acc_plot.legend(loc=4)
    #plt.tight_layout()
    #plt.show()

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

```
Training...

EPOCH 1 ...
Training Accuracy = 0.870,
Validation Accuracy = 0.807

EPOCH 2 ...
Training Accuracy = 0.923,
Validation Accuracy = 0.882

EPOCH 3 ...
Training Accuracy = 0.954,
Validation Accuracy = 0.903

EPOCH 4 ...
Training Accuracy = 0.963,
Validation Accuracy = 0.921

EPOCH 5 ...
Training Accuracy = 0.970,
Validation Accuracy = 0.927

EPOCH 6 ...
Training Accuracy = 0.972,
Validation Accuracy = 0.929

EPOCH 7 ...
Training Accuracy = 0.980,
Validation Accuracy = 0.941

EPOCH 8 ...
Training Accuracy = 0.974,
Validation Accuracy = 0.942

EPOCH 9 ...
Training Accuracy = 0.984,
Validation Accuracy = 0.946

EPOCH 10 ...
Training Accuracy = 0.986,
Validation Accuracy = 0.943

EPOCH 11 ...
```

```
Training Accuracy = 0.987,
Validation Accuracy = 0.940

EPOCH 12 ...
Training Accuracy = 0.984,
Validation Accuracy = 0.941

EPOCH 13 ...
Training Accuracy = 0.991,
Validation Accuracy = 0.956

EPOCH 14 ...
Training Accuracy = 0.988,
Validation Accuracy = 0.957

EPOCH 15 ...
Training Accuracy = 0.990,
Validation Accuracy = 0.949

EPOCH 16 ...
Training Accuracy = 0.987,
Validation Accuracy = 0.948

EPOCH 17 ...
Training Accuracy = 0.993,
Validation Accuracy = 0.950

EPOCH 18 ...
Training Accuracy = 0.992,
Validation Accuracy = 0.949

EPOCH 19 ...
Training Accuracy = 0.991,
Validation Accuracy = 0.952

EPOCH 20 ...
Training Accuracy = 0.992,
Validation Accuracy = 0.947

Model saved
```

In [33]:
```
# LOSS PLOTS
loss_plot = plt.subplot(1,1,1)
loss_plot.set_title('Loss')
loss_plot.plot(train_loss_history, 'r', label='Training Loss')
loss_plot.plot(validation_loss_history, 'b', label='Validation Loss')
loss_plot.set_xlim([0, EPOCHS])
loss_plot.legend(loc=4)

plt.savefig('./examples/plot3.png')
```



In [35]:
```
# EVALUATE ACCURACY on test data
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))

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

```
Test Accuracy = 0.932
```

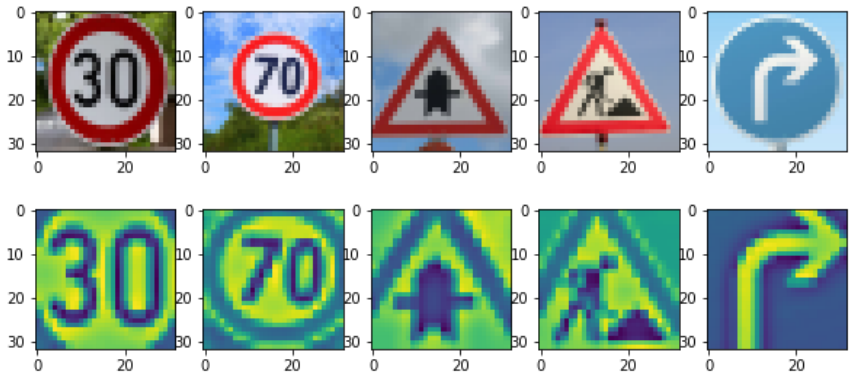
Step 3: Test a Model on New Images

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 [36]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.

import glob
#import matplotlib.image as mpimg

# Test on new images from the internet
X_new_test = []
for filename in glob.glob('./examples/test_*.png'):
    X_new_test.append(cv2.imread(filename))

# Make sure the input images are the same size as training set
X_new_test = [resizeImage(image) for image in X_new_test]


fig1 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(cv2.cvtColor(X_new_test[j],cv2.COLOR_BGR2RGB))

fig2 = plt.figure(figsize=(10,10))
for j in range(0,5):
    plt.subplot(1,5,j+1)
    plt.imshow(pipeline(X_new_test[j]))

X_test_new = np.zeros((len(X_new_test),32,32))
# Create labels
for i in range(0,5):
    X_test_new[i] = pipeline(X_new_test[i])

X_new_test = X_test_new[..., newaxis]
y_new_test = [1,4,11,25,33]
```



Predict the Sign Type for Each Image

In [37]:
```
### Run the predictions here and use the model to output the prediction for each image.
### Make sure to pre-process the images with the same pre-processing pipeline used earlier.
### Feel free to use as many code cells as needed.


with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predictions,topk = predict(X_new_test, y_new_test)
    print("Labels: ", y_new_test)
    print("Predictions: ",predictions)
```

```
Labels:  [1, 4, 11, 25, 33]
Predictions:  [ 1  0 11 25 33]
```

Analyze Performance

In [38]:
```
### Calculate the accuracy for these 5 new images.
### For example, if the model predicted 1 out of 5 signs correctly, it's 20% accurate on these new images.
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    loss, accuracy = evaluate(X_new_test, y_new_test, sess)
    print("Accuracy = {:.3f}".format(accuracy))
    # print()
    # print(topk)
```

```
Accuracy = 0.800
```

Output Top 5 Softmax Probabilities For Each Image Found on the Web

For each of the new images, print out the model's softmax probabilities to show the **certainty** of the model's predictions (limit the output to the top 5 probabilities for each image). tf.nn.top_k (https://www.tensorflow.org/versions/r0.12/api_docs/python/nn.html#top_k) could prove helpful here.

The example below demonstrates how tf.nn.top_k can be used to find the top k predictions for each image.

`tf.nn.top_k` will return the values and indices (class ids) of the top k predictions. So if k=3, for each sign, it'll return the 3 largest probabilities (out of a possible 43) and the correspoding class ids.

Take this numpy array as an example. The values in the array represent predictions. The array contains softmax probabilities for five candidate images with six possible classes. `tk.nn.top_k` is used to choose the three classes with the highest probability:

```
# (5, 6) array
a = np.array([[ 0.24879643,  0.07032244,  0.12641572,  0.34763842,  0.07893497,
         0.12789202],
       [ 0.28086119,  0.27569815,  0.08594638,  0.0178669 ,  0.18063401,
         0.15899337],
       [ 0.26076848,  0.23664738,  0.08020603,  0.07001922,  0.1134371 ,
         0.23892179],
       [ 0.11943333,  0.29198961,  0.02605103,  0.26234032,  0.1351348 ,
         0.16505091],
       [ 0.09561176,  0.34396535,  0.0643941 ,  0.16240774,  0.24206137,
         0.09155967]])
```

Running it through `sess.run(tf.nn.top_k(tf.constant(a), k=3))` produces:

```
TopKV2(values=array([[ 0.34763842,  0.24879643,  0.12789202],
       [ 0.28086119,  0.27569815,  0.18063401],
       [ 0.26076848,  0.23892179,  0.23664738],
       [ 0.29198961,  0.26234032,  0.16505091],
       [ 0.34396535,  0.24206137,  0.16240774]]), indices=array([[3, 0, 5],
       [0, 1, 4],
       [0, 5, 1],
       [1, 3, 5],
       [1, 4, 3]], dtype=int32))
```

Looking just at the first row we get `[ 0.34763842,  0.24879643,  0.12789202]`, you can confirm these are the 3 largest probabilities in `a`. You'll also notice `[3, 0, 5]` are the corresponding indices.

In [39]:
```
### Print out the top five softmax probabilities for the predictions on the German traffic sign images fou
nd on the web.
### Feel free to use as many code cells as needed.

with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('.'))
    predictions,topk = predict(X_new_test, y_new_test)
    #print("Accuracy = {:.3f}".format(accuracy))
    #float_formatter = lambda x: "%.3f" % x
    np.set_printoptions(precision=4, suppress=True)
    print(topk)
```

```
TopKV2(values=array([[ 0.9974,  0.0018,  0.0003,  0.0003,  0.0001],
       [ 0.3748,  0.347 ,  0.0978,  0.0886,  0.0327],
       [ 0.9985,  0.0013,  0.0001,  0.    ,  0.    ],
       [ 0.9997,  0.0001,  0.0001,  0.    ,  0.    ],
       [ 0.9689,  0.0143,  0.0033,  0.0031,  0.002 ]], dtype=float32), indices=array([[ 1,  5, 31, 18, 2
6],
       [ 0,  1,  4, 29,  5],
       [11, 30, 40, 26, 21],
```

```
[25, 30, 11, 35, 26],
[33, 35, 11, 26,  5]], dtype=int32))
```

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.

In [16]:
```
### 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 calc
ulated state of a specific weight layer
```