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

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

training_file = "traffic-signs-data/train.p"
validation_file="traffic-signs-data/valid.p"
testing_file = "traffic-signs-data/test.p"

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

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

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 [2]:
```
### Replace each question mark with the appropriate value.
### Use python, pandas or numpy methods rather than hard coding the re
sults
import numpy as np

# TODO: Number of training examples
n_train = len(X_train)

# TODO: Number of validation examples
n_validation = len(X_valid)

# TODO: Number of testing examples.
n_test = len(X_test)

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

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

print("Number of training examples =", n_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 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

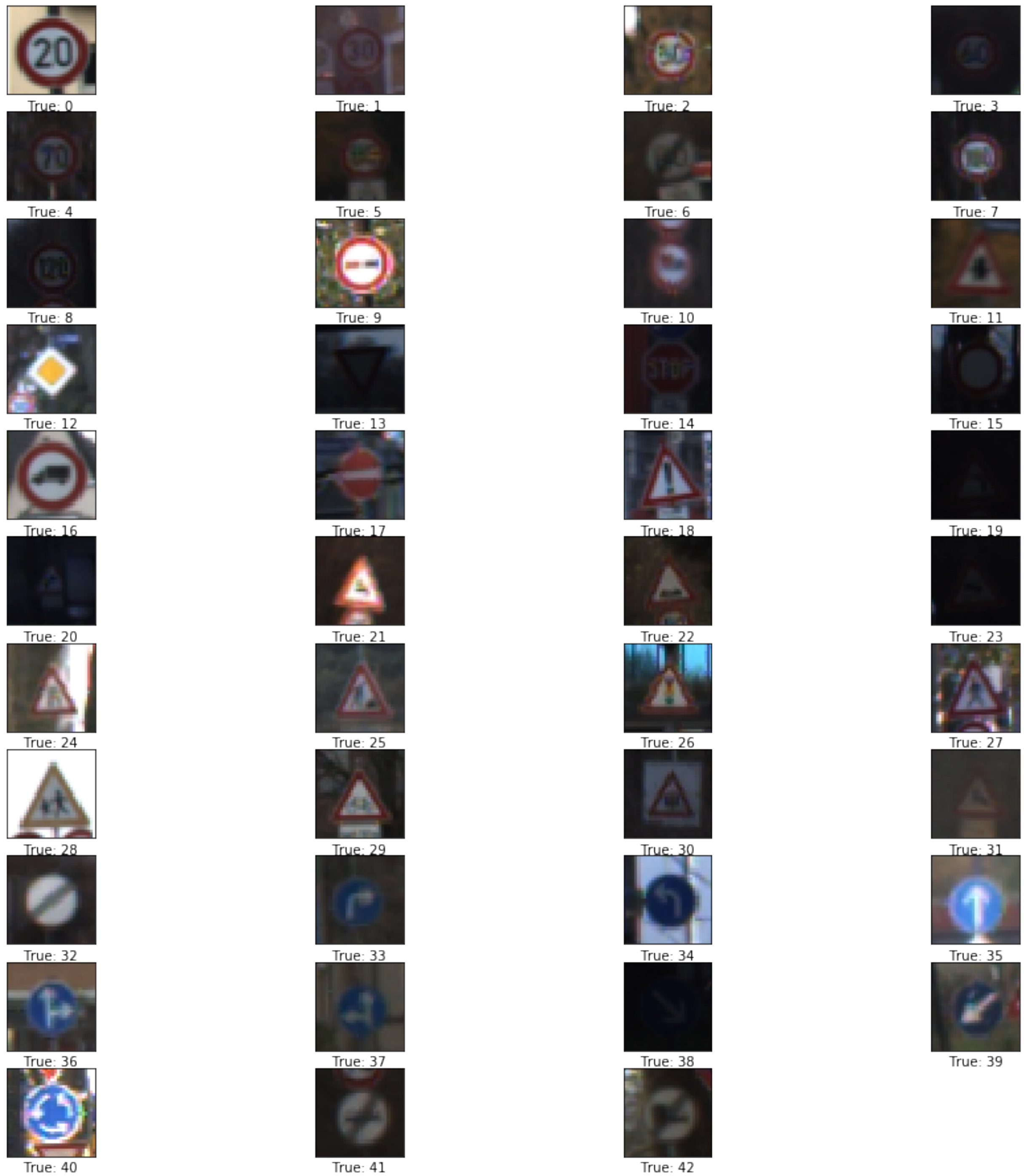
Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

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.

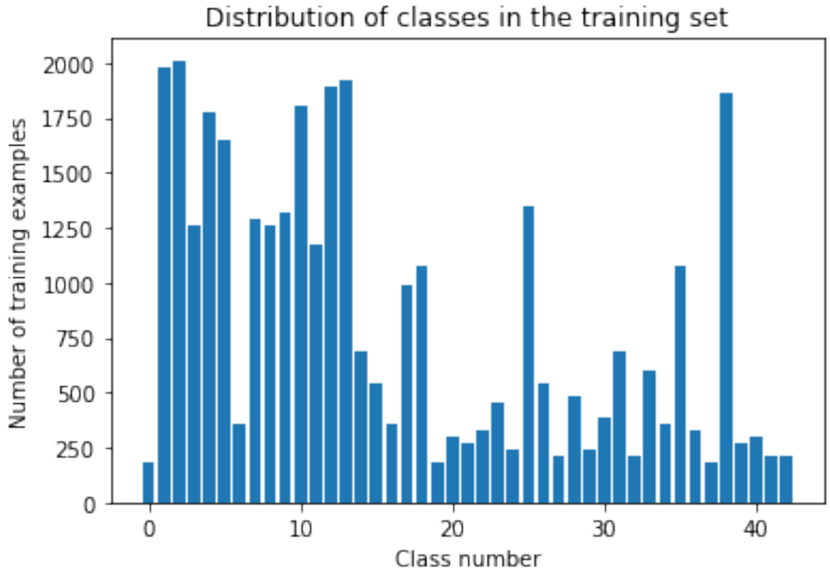
NOTE: It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections. It can be interesting to look at the distribution of classes in the training, validation and test set. Is the distribution the same? Are there more examples of some classes than others?

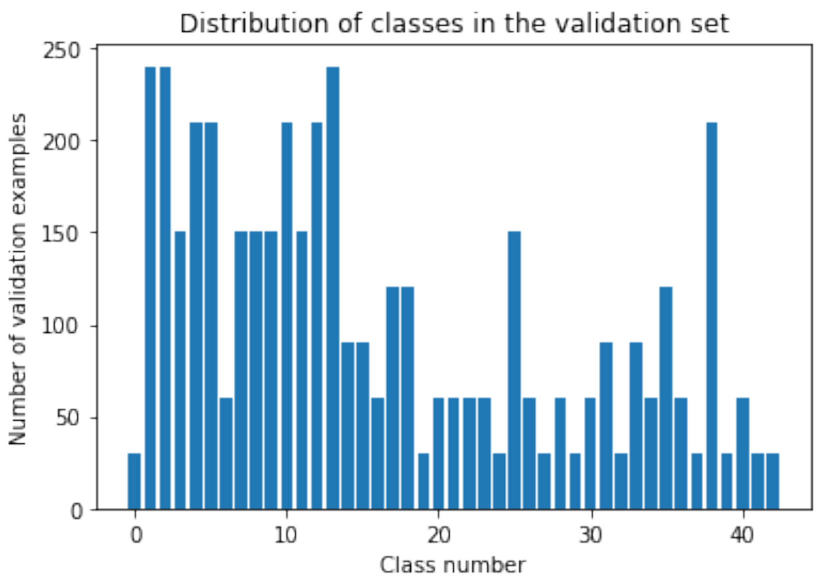
In [3]:
```
### Data exploration visualization code goes here.
### Feel free to use as many code cells as needed.
import math
import matplotlib.pyplot as plt
%matplotlib inline

def plot_images(images, class_true, class_pred):
    assert len(images) == len(class_true)

    n_cols = 4
    n_rows = math.ceil(len(images)/4)
    fig = plt.figure(figsize=(16, 16))

    for i in range(len(images)):
        ax = fig.add_subplot(n_rows,n_cols,i+1)
        plt.imshow(images[i])
        if class_pred is None:
             xlabel = "True: {0}".format(class_true[i])
        else:
             xlabel = "True: {0}, Pred: {1}".format(class_true[i], cla
ss_pred[i])

        ax.set_xlabel(xlabel)

        #Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])
```

In [4]:
```
images = []
true_class = np.unique(y_train)
for i in range(len(true_class)):
    X_temp = X_train[y_train == i]
    images.append(X_temp[0])

plot_images(images=images, class_true=true_class, class_pred = None)
plt.savefig('Examples/Visualization_1')
```


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

In [5]:
```
## Checking the distribution of the classes in the training set and va
lidation set
dict_num_class = dict(zip(np.unique(y_train),np.zeros(len(np.unique(y_
train)))))

for i in y_train:
    dict_num_class[i]+=1

plt.bar(list(dict_num_class.keys()), list(dict_num_class.values()))
plt.title("Distribution of classes in the training set")
plt.xlabel("Class number")
plt.ylabel("Number of training examples")
plt.show()

dict_num_class_valid = dict(zip(np.unique(y_valid),np.zeros(len(np.uni
que(y_valid)))))

for i in y_valid:
    dict_num_class_valid[i]+=1

plt.bar(list(dict_num_class_valid.keys()), list(dict_num_class_valid.v
alues()))
plt.title("Distribution of classes in the validation set")
plt.xlabel("Class number")
plt.ylabel("Number of validation examples")
plt.show()
```


Distribution of classes in the training set
Number of training examples
Class number
0
250
500
750
1000
1250
1500
1750
2000
0
10
20
30
40


Distribution of classes in the validation set
Number of validation examples
Class number
0
50
100
150
200
250
0
10
20
30
40

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 [6]:
```
### Preprocess the data here. It is required to normalize the data. Ot
her preprocessing steps could include
### converting to grayscale, etc.
### Feel free to use as many code cells as needed.
```

In [7]:
```
import cv2
def slicing(image):
    return image[3:29,3:29,:]

def blur(image):
    image_new = cv2.GaussianBlur(image,(7,7),10)
    return cv2.addWeighted(image,2.3, image_new, -1, 0)

def translate(image):
    M = np.float32([[1,0,3],[0,1,2]])
    return cv2.warpAffine(image,M,(image.shape[1],image.shape[0]))

def rotate(image):
    M = cv2.getRotationMatrix2D((image.shape[1]/2,image.shape[0]/2),np
.random.rand(1)*10,1)
    return cv2.warpAffine(image,M,(image.shape[1],image.shape[0]))

def distort(image):
    image1 = slicing(image)
    image2 = blur(image1)
    image3 = rotate(image2)
    return image3
```

In [8]:
```
### Showing the difference between the raw image and the preprocessed image
example_image = 4900
image_processed = blur(slicing(X_train[example_image]))

plt.figure(figsize=(6, 4))
plt.subplot(121)
plt.xlabel('Raw Image')
plt.xticks(())
plt.yticks(())
plt.imshow(X_train[example_image])
plt.subplot(122)
plt.xlabel('Processed Image')
plt.xticks(())
plt.yticks(())
plt.imshow(image_processed)
plt.savefig('Examples/Raw-and-processed-Images')
```

In [9]:
```
X_train_new = []
y_train_new = []

for i in range(n_train):
    X_train_new.append(blur(slicing(X_train[i])))
    y_train_new.append(y_train[i])

for i in range(n_validation):
    X_train_new.append(blur(slicing(X_valid[i])))
    y_train_new.append(y_valid[i])

desired_length = 1500

for i in range(n_classes):
        cnt = 0
        X_train_less = X_train[y_train == i]
        current_length = len(X_train_less)
        add_length = desired_length - current_length
        while (cnt <= add_length):
            ind = np.random.randint(0,current_length-1)
            X_train_new.append(distort(X_train_less[ind]))
            y_train_new.append(i)
            cnt = cnt + 1
```

In [10]:
```
from sklearn.model_selection import train_test_split
X_train_p, X_valid_p, y_train_p, y_valid_p = train_test_split(X_train_
new,y_train_new, test_size = 0.15,random_state=0)
```

In [11]:
```
X_train_p[0].shape
len(X_train_p)
```

Out[11]: 61051

Model Architecture

In [12]:
```
### Define your architecture here.
import tensorflow as tf
from tensorflow.contrib.layers import flatten

keep_prob = tf.placeholder(tf.float32) # probability to keep units

def LeNet(x):
    # Arguments used for tf.truncated_normal, randomly defines variabl
es for the weights and biases for each layer
    mu = 0
    sigma = 0.01
```

```
    # TODO: Layer 1: Convolutional. Input = 26x26x3. Output = 22x22x6.
    W_layer_1 = tf.Variable(tf.random_normal([5, 5, 3, 6],mu,sigma))
    Bias_layer_1 = tf.Variable(tf.random_normal([6],mu,sigma))
    conv_layer_1 = tf.nn.bias_add(tf.nn.conv2d(x, W_layer_1, strides=[
1, 1, 1, 1], padding='VALID'),Bias_layer_1)

    # TODO: Activation.
    conv_layer_1 = tf.nn.relu(conv_layer_1)

    # TODO: Pooling. Input = 22x22x6. Output = 11x11x6.
    layer_1 = tf.nn.max_pool(conv_layer_1, ksize=[1, 2, 2, 1], strides
=[1, 2, 2, 1],padding='SAME')

    # TODO: Layer 2: Convolutional. Output = 7x7x16.
    W_layer_2 = tf.Variable(tf.random_normal([5, 5, 6, 16],mu,sigma))
    Bias_layer_2 = tf.Variable(tf.random_normal([16],mu,sigma))
    conv_layer_2 = tf.nn.bias_add(tf.nn.conv2d(layer_1, W_layer_2, str
ides=[1, 1, 1, 1], padding='VALID'),Bias_layer_2)

    # TODO: Activation.
    conv_layer_2 = tf.nn.relu(conv_layer_2)

    # TODO: Pooling. Input = 7x7x16. Output = 4x4x16.
    layer_2 = tf.nn.max_pool(conv_layer_2, ksize=[1, 2, 2, 1], strides
=[1, 2, 2, 1],padding='SAME')

    # TODO: Flatten. Input = 4x4x16. Output = 256.
    layer_2 = flatten(layer_2)

    # TODO: Layer 3: Fully Connected. Input = 256. Output = 120.
    W_layer_3 = tf.Variable(tf.random_normal([256,120],mu,sigma))
    Bias_layer_3 = tf.Variable(tf.random_normal([120],mu,sigma)),
    layer_3 = tf.add(tf.matmul(layer_2,W_layer_3),Bias_layer_3)

    # TODO: Activation.
    layer_3 = tf.nn.relu(layer_3)
    layer_3= tf.nn.dropout(layer_3, keep_prob)

    # TODO: Layer 4: Fully Connected. Input = 120. Output = 84.
    W_layer_4 = tf.Variable(tf.random_normal([120,84],mu,sigma))
    Bias_layer_4 = tf.Variable(tf.random_normal([84],mu,sigma)),
    layer_4 = tf.add(tf.matmul(layer_3,W_layer_4),Bias_layer_4)

    # TODO: Activation.
    layer_4 = tf.nn.relu(layer_4)
    layer_4= tf.nn.dropout(layer_4, keep_prob)

    # TODO: Layer 5: Fully Connected. Input = 84. Output = 43.
    W_layer_5 = tf.Variable(tf.random_normal([84,43],mu,sigma))
```

```
    Bias_layer_5 = tf.Variable(tf.random_normal([43],mu,sigma)),
    logits = tf.add(tf.matmul(layer_4,W_layer_5),Bias_layer_5)

    return logits
### Feel free to use as many code cells as needed.
```

In [13]:
```
x = tf.placeholder(tf.float32, (None, 26, 26, 3))
y = tf.placeholder(tf.int32, (None))
one_hot_y = tf.one_hot(y, 43)
```

In [14]:
```
rate = 0.001

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)
```

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 [15]:
```
### Train your model here.
### Calculate and report the accuracy on the training and validation s
et.
### 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.
```

In [16]:
```
correct_prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(one_hot_
y, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_prediction, tf.flo
at32))
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[of
fset:offset+BATCH_SIZE]
        accuracy = sess.run(accuracy_operation, feed_dict={x: batch_x,
y: batch_y, keep_prob:1})
        total_accuracy += (accuracy * len(batch_x))
    return total_accuracy / num_examples
```

In [17]:
```
from sklearn.utils import shuffle

EPOCHS = 15
BATCH_SIZE = 128

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

    print("Training...")
    print()
    for i in range(EPOCHS):
        print("EPOCH {} ...".format(i+1))
        X_train_p, y_train_p = shuffle(X_train_p, y_train_p)
        for offset in range(0, num_examples, BATCH_SIZE):
            end = offset + BATCH_SIZE
            batch_x, batch_y = X_train_p[offset:end], y_train_p[offset
:end]
            sess.run(training_operation, feed_dict={x: batch_x, y: bat
ch_y, keep_prob: 0.6})
            if (offset% 25600 == 0):
                training_accuracy = evaluate(batch_x, batch_y)
                print("Training Accuracy = {:.3f}".format(training_acc
uracy),"offset = {}".format(offset))

        validation_accuracy = evaluate(X_valid_p, y_valid_p)
        print("Validation Accuracy = {:.3f}".format(validation_accurac
y))
        print()

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

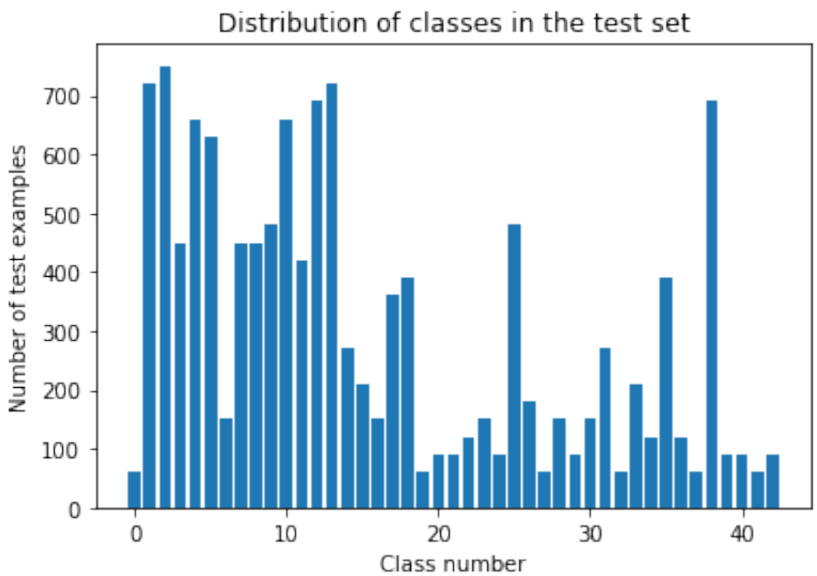
```
Training...

EPOCH 1 ...
Training Accuracy = 0.031 offset = 0
Training Accuracy = 0.195 offset = 25600
Training Accuracy = 0.523 offset = 51200
Validation Accuracy = 0.585

EPOCH 2 ...
Training Accuracy = 0.602 offset = 0
Training Accuracy = 0.750 offset = 25600
Training Accuracy = 0.914 offset = 51200
Validation Accuracy = 0.873

EPOCH 3 ...
Training Accuracy = 0.844 offset = 0
```

```
Training Accuracy = 0.945 offset = 25600
Training Accuracy = 0.914 offset = 51200
Validation Accuracy = 0.938

EPOCH 4 ...
Training Accuracy = 0.961 offset = 0
Training Accuracy = 0.906 offset = 25600
Training Accuracy = 0.977 offset = 51200
Validation Accuracy = 0.957

EPOCH 5 ...
Training Accuracy = 0.969 offset = 0
Training Accuracy = 0.953 offset = 25600
Training Accuracy = 0.969 offset = 51200
Validation Accuracy = 0.961

EPOCH 6 ...
Training Accuracy = 0.984 offset = 0
Training Accuracy = 0.945 offset = 25600
Training Accuracy = 0.961 offset = 51200
Validation Accuracy = 0.971

EPOCH 7 ...
Training Accuracy = 0.992 offset = 0
Training Accuracy = 0.984 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.974

EPOCH 8 ...
Training Accuracy = 0.984 offset = 0
Training Accuracy = 1.000 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.974

EPOCH 9 ...
Training Accuracy = 0.992 offset = 0
Training Accuracy = 0.984 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.973

EPOCH 10 ...
Training Accuracy = 0.961 offset = 0
Training Accuracy = 0.961 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.978

EPOCH 11 ...
Training Accuracy = 0.992 offset = 0
Training Accuracy = 1.000 offset = 25600
Training Accuracy = 0.984 offset = 51200
```

```
Validation Accuracy = 0.978

EPOCH 12 ...
Training Accuracy = 0.984 offset = 0
Training Accuracy = 0.961 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.980

EPOCH 13 ...
Training Accuracy = 1.000 offset = 0
Training Accuracy = 1.000 offset = 25600
Training Accuracy = 0.984 offset = 51200
Validation Accuracy = 0.979

EPOCH 14 ...
Training Accuracy = 0.984 offset = 0
Training Accuracy = 1.000 offset = 25600
Training Accuracy = 0.992 offset = 51200
Validation Accuracy = 0.982

EPOCH 15 ...
Training Accuracy = 0.992 offset = 0
Training Accuracy = 0.969 offset = 25600
Training Accuracy = 0.984 offset = 51200
Validation Accuracy = 0.981

Model saved
```

Accuracy on the given test set

In [18]:
```
dict_num_class_test = dict(zip(np.unique(y_test),np.zeros(len(np.uniqu
e(y_test)))))

for i in y_test:
    dict_num_class_test[i]+=1

plt.bar(list(dict_num_class_test.keys()), list(dict_num_class_test.val
ues()))
plt.title("Distribution of classes in the test set")
plt.xlabel("Class number")
plt.ylabel("Number of test examples")
plt.show()
```

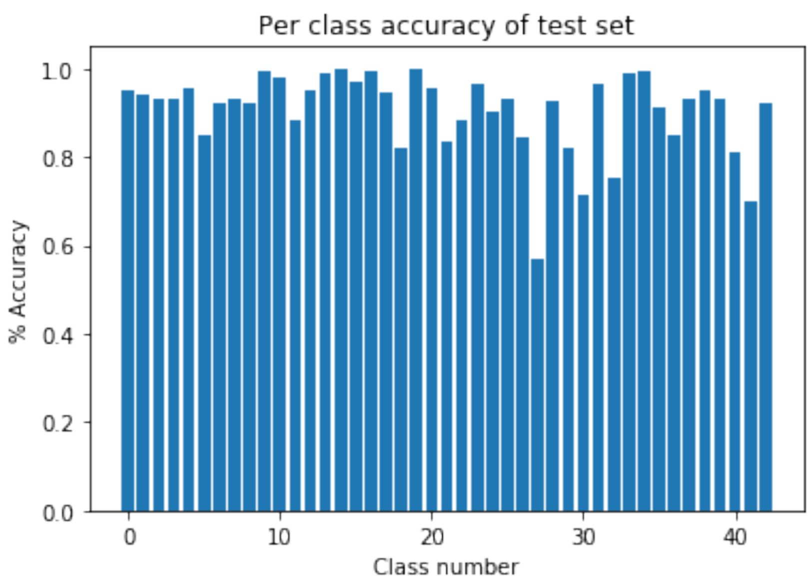


In [19]:
```
X_test_alt = []

for i in range(n_test):
    X_test_alt.append(blur(slicing(X_test[i])))

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

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

```
Test Accuracy = 0.928
```

In [20]:

```
## Plotting accuracy for each of the classes:
dict_class_accuracy = dict(zip(np.unique(y_test),np.zeros(len(np.uniqu
e(y_test)))))
X_test_alt_np = np.asarray(X_test_alt)

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

    for i in range(n_classes):
        X_particular_class = X_test_alt_np[y_test == i]
        y_particular_class = i*np.ones(len(X_particular_class))
        dict_class_accuracy[i] = evaluate(X_particular_class, y_partic
ular_class)


plt.bar(list(dict_class_accuracy.keys()), list(dict_class_accuracy.val
ues()))
plt.title("Per class accuracy of test set")
plt.xlabel("Class number")
plt.ylabel("% Accuracy")
plt.show()
```



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 [21]:
```
### Load the images and plot them here.
### Feel free to use as many code cells as needed.
import os
def mylistdir(directory):
    """A specialized version of os.listdir() that ignores files that
    start with a leading period."""
    filelist = os.listdir(directory)
    return [x for x in filelist
            if not (x.startswith('.'))]

list_images = mylistdir("New_Test_images/")
len_list_images = len(list_images)
#os.mkdir('test_images_output')

X_test_raw = []
y_test_raw = [3,9,35,13,14]
for i,im in zip(range(len_list_images),list_images):
    image = plt.imread(os.path.join("New_Test_images/",im))
    image = cv2.resize(image, (32, 32))
    X_test_raw.append(image)
```

In [22]:
```
# Processing images
X_test_new = []
y_test_new = [3,9,35,13,14]
for i in range(len(X_test_raw)):
    image_in = blur(slicing(X_test_raw[i]))
    X_test_new.append(image_in)
    plt.axis('off')
    image_out = plt.imshow(image_in)
    plt.savefig('examples/Test_'+str(i+1))
```

Predict the Sign Type for Each Image

In [23]:
```
### Run the predictions here and use the model to output the predictio
n for each image.
### Make sure to pre-process the images with the same pre-processing p
ipeline 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 = sess.run(tf.argmax(logits,1), feed_dict={x: X_test_n
ew, keep_prob:1})
print("Class predictions={}".format(predictions))
```

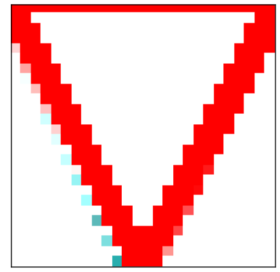
```
Class predictions=[ 3  9 35 13 14]
```

In [24]:
```
plot_images(images=X_test_new, class_true=y_test_new, class_pred = pre
dictions)
```

True: 3, Pred: 3


True: 9, Pred: 9


True: 35, Pred: 35


True: 13, Pred: 13

True: 14, Pred: 14

Analyze Performance

In [25]:
```
### 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('.'))

    test_new_accuracy = evaluate(X_test_new, y_test_new)
    print("Test Accuracy on the new images = {:.3f}".format(test_new_a
ccuracy))
```

```
Test Accuracy on the new images = 1.000
```

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

In [26]:
```
### Print out the top five softmax probabilities for the predictions o
n the German traffic sign images found 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('.'))
    probabilities = sess.run(tf.nn.softmax(logits), feed_dict={x: X_te
st_new, keep_prob:1})
    top_k_vals = sess.run(tf.nn.top_k(tf.constant(probabilities), k=5)
)
top_k_vals
```

Out[26]:
```
TopKV2(values=array([[  9.98250186e-01,   1.17269345e-03,   5.249806
56e-04,
          4.50356129e-05,   1.97974214e-06],
       [  9.99998808e-01,   1.17700267e-06,   1.26837456e-08,
          7.27348581e-09,   1.36813416e-09],
       [  7.11053848e-01,   1.26245901e-01,   1.11906268e-01,
          3.01445890e-02,   1.38063980e-02],
       [  1.00000000e+00,   5.69038506e-21,   2.80684414e-26,
          4.01137850e-27,   5.21360380e-31],
       [  9.99998093e-01,   1.00458021e-06,   3.57222575e-07,
          2.46024541e-07,   2.01043505e-07]], dtype=float32), indice
s=array([[ 3,  5,  2, 16, 32],
       [ 9, 10, 13, 16, 41],
       [35, 34, 36, 28, 25],
       [13, 14, 10,  9,  1],
       [14, 17,  2, 13,  1]], dtype=int32))
```