Traffic Sign Recognition

Build a Traffic Sign Recognition Project

The goals / steps of this project are the following:
- *Load the data set (see below for links to the project data set)*
- *Explore, summarize and visualize the data set*
- *Design, train and test a model architecture*
- *Use the model to make predictions on new images*
- *Analyze the softmax probabilities of the new images*
- *Summarize the results with a written report*

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup

You're reading it! and here is a link to my project code

Data Set Summary & Exploration

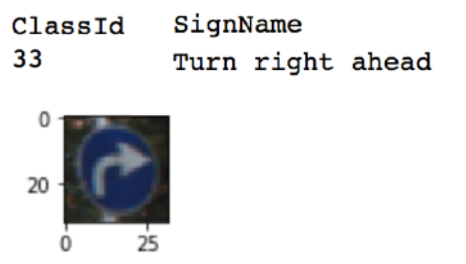
1. Provide a basic summary of the data set.

I used the pandas library to calculate summery statistics of the traffic signs data set and numpy methods to compute the number of unique 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
```

2. Include an exploratory visualisation of the dataset.

The dataset include coloured 32x32 images of German traffic signs. The statistics of the set is shown above. The set is supplied with an CSV file that includes an explanation of the associated to the traffic signs 43 labels. Here is an image of a random sample with the associated from the CSV file meaning:



```
ClassId   SignName
33        Turn right ahead
```

Design and Test a Model Architecture

1. Describe how you preprocessed the image data.

I tried several methods of preprocessing the data. For the first, I used 3 channels coloured shuffled images that I normalised using mean and standard deviation. The reason for that is because in the process of training our network, we are going to be multiplying (weights) and adding to (biases) these initial inputs in order to cause activations that we then backpropogate with the gradients to train the model. We'd like in this process for each feature to have a similar range so that our gradients don't go out of control.

image_norm = (image - mean) / std

After training and testing the model, I tied also converting the images into grayscale still using the same input normalisation method. Converted to grayscale images look like that:

```
new shape  (32, 32, 1)
(34799, 32, 32, 1)
```

2. Describe what your final model architecture looks like including model type, layers, layer sizes, connectivity, etc.).

My final model consisted of the following layers:

Layer	Description
Input	32x32x3 RGB image or 32x32x1 Grayscale image
Convolution 5x5	1x1 stride, valid padding, outputs 28x28x6
RELU	
Max pooling	2x2stride, valid padding, 14x4x6 outputs
Convolution 5x5	1x1 stride, valid padding, outputs 10x10x16
RELU	
Max pooling	2x2stride, valid padding, 5x5x16 outputs
Flatten	400 outputs
Fully connected	120 outputs
RELU	
Dropout	75 %
Fully connected	84 outputs
RELU	
Dropout	75 %

Layer	Description
Fully connected	43 outputs

3. Describe how you trained your model.

To train the model, I used batch size of 128, learning rate 0.001 and running 50 epochs. The data was already split into training, validation, and testing sets.
A more faster Adam method of stochastic optimisation was used which lead to relatively quick and reliable convergence.

4. Describe the approach taken for finding a solution and getting the validation set accuracy to be at least 0.93.

I started with the original LeNet architecture and just adapted it to be used for RGB images. Input shuffling was already implemented and with the first run of 10 Epochs I've got already pretty decent result. Lowing the learning rate I figured out was not leading the the accuracy improvement but increasing the number of Epochs did. To avoid memorising the training set by the network I added 75 percent dropout on the two last fully connected layers. I experimented with image pre-processing and grayscale conversion. Learning Python and dealing with multiple Python libraries problems was in fact more iterative process that left nearly not time to concentrate on the model tuning. Although the model showed a very good performance on the training and validation sets, it performed slightly worse on the test set and nearly failed on the images I downloaded from the Internet. If I would have time to come back to this exciting expertise I would like to add noise to the training set to reflect the grains and artefacts that were present after downscaling the downloaded images. Also the fact that the image ratio was not squared, it looks like padding has also some influence on the way the model performs since the training set didn't have image paddings. I would like to add random shifts or maybe some perspective distortions (if I only new how to do it easily in Python :-)) to the training set. In my understanding this would be the minimum preprocessing required. The next steps would then involve some architectural tuning. My final results are the following:

```
Training Accuracy = 0.994
Validation Accuracy = 0.946
Test Accuracy = 0.911

Test on 9 downloaded from web images:
Test Accuracy = 22.2%
```

LeNet architecture in my opinion is a good hybrid architecture that takes advantage of both Convolution Neural Network and Deep Neural Networks. The convolutional part helps to extract and combine meaningful features when the deep part helps to separate the data correctly.

Test a Model on New Images

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.

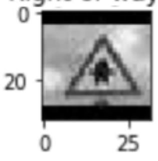
The accuracy and reflection on the problem are described in the previous section. Here I will provide the final results.
As it can be seen from the final output, only two images are correctly classified. Please note, that the image quality is rather poor comparing to the training set.

```
Testing 9 images

predictions:
[11 42  1  9 37  2 38 25  1]
```

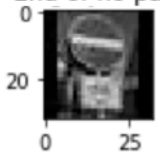
1

```
ClassId   SignName
  11      Right-of-way at the next intersection
```



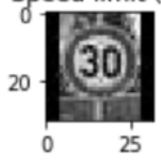
2

```
ClassId   SignName
  42      End of no passing by vehicles over 3.5 metric ...
```



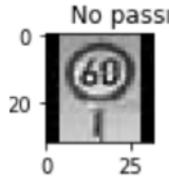
3

```
ClassId   SignName
  1       Speed limit (30km/h)
```



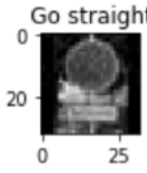
4

```
ClassId   SignName
  9            No passing
```



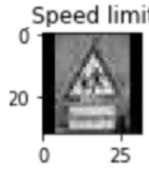
5

```
ClassId        SignName
  37      Go straight or left
```



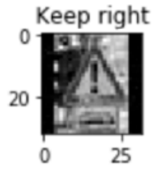
6

```
ClassId        SignName
  2       Speed limit (50km/h)
```



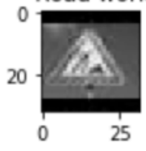
7

```
ClassId   SignName
  38      Keep right
```



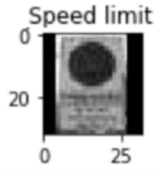
8

```
ClassId   SignName
  25      Road work
```



9

```
ClassId        SignName
  1       Speed limit (30km/h)
```



2. Discuss the model's predictions on these new traffic signs and compare the results to predicting on the test set.

Additionally to testing the model on the complete test set, I ran the model on randomly selected 5 imaged from the test dataset and displayed the results. To recall, the test accuracy on the complete set was 91.1% which is slightly lower than on the validation set (94.6%). The result on the randomly selected 5 test images form the test dataset was mostly 5 out 5 (sometimes 4 out 5) which is 100% (or 80% respectively). When running on the downloaded from the web 9 images, the results were rather poor ending up with 2 out 9 images accuracy (22.2%) - thus the model is overfitting.

The code for making predictions on my final model is located in the 17th cell of the Ipython notebook. I used *tf.argmax* to compute predictions and *tf.nn.softmax* to compute probabilities. The output above is based on *tf.argmax* predictions.

Out of top 5 probabilities computed with *tf.nn.softmax* the believe of the image class looks like the following (not rounded, just cut to 2 digits after comma):

Image	Probability	Prediction	Valid
1	0.99	11 - Right-of-way at the next intersection	No
2	0.97	12 - Priority road	No
3	0.99	1 - Speed limit (30km/h)	Yes
4	0.99	40 - Roundabout mandatory	No
5	0.99	39 - Keep left	No
6	0.97	31 - Wild animals crossing	No
7	0.65	38 - Keep right	No
8	0.80	25 - Road work	Yes
9	0.96	17 - No entry	No

As it can be seen, the actual accuracy is the same, but incorrect predictions slightly vary. The other four probabilities in every case are extremely low. But the image quality is at the edge of human understandability, so I am anyway impressed that the network can perform at all on such a dataset.

Visualizing the Neural Network

Here I would like to receive support since I was not able to visualise it and couldn't find examples in internet. I would kindly appreciated if you could provide me the solution code or pointed me what I am doing wrong and how should it be done.