Traffic Sign Recognition

Report

By Roy Veshovda

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 / README

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

Data Set Summary & Exploration

The code for this step is contained in the 3rd and 4th code cells of the Jupyter notebook.

I used the pandas library to calculate summary statistics of the traffic signs data set:

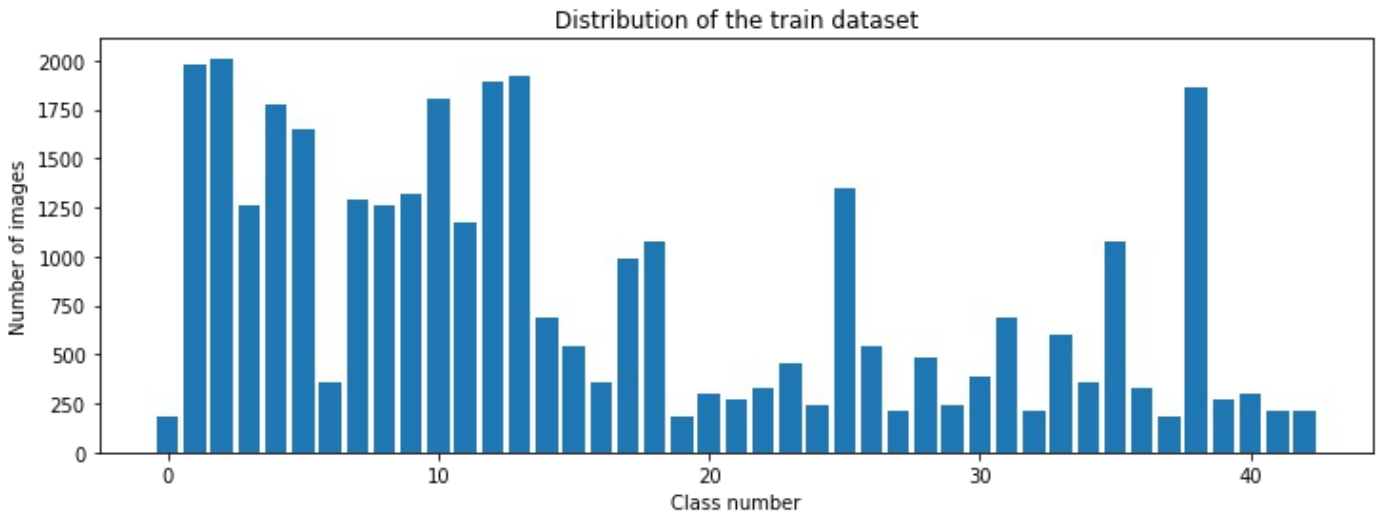
- The size of training set is 34799
- The size of validation set is 4410
- The size of test set is 12630
- The shape of a traffic sign image is 32x32x3
- The number of unique classes/labels in the data set is 43

Details

The code for this step is contained in the 4th code cell of the Jupyter notebook.

First I show a sample of an image from each of the 43 classes. Below those I show a bar chart of how many images of each class exists in the training data.

That bar chart is also shown below.



Design and Test a Model Architecture

1. Preprocessing

The code for this step is contained in the 5th code cell of the Jupyter notebook.

My intuition was to keep the color channels, as I believe the signs will be easier to separate if colors are also processed by the network.

But I wanted to normalize the color channels between -1.0 and 1.0 (instead of the original 0-255). Normalized values (in combination with non-zero initialized weights) are important for the backpropagation algorithm to operate optimal.

I also am running a shuffle of the training data.

The labels are one-hot-coded, meaning they are represented as a bit-string with one bit representing each class. For examle the class number 22, is translated to a bit-string with a 1 in position 22, and zeros before and after. This is also to let network be easier to train.

2. Training, validation and testing data.

The data came ready split into train, validation and test arrays, so no more pre-processing was needed here.

3. Model architecture

The code for my final model is located in the 6th cell of the Jupyter notebook.

My final model consisted of the following layers:

Layer	Description
Input	32x32x3 RGB image
1: Convolution 5x5	1x1 stride, valid padding, outputs 28x28x16
1: RELU	
1: Max pooling	2x2 stride, outputs 14x14x16
2: Convolution 5x5	1x1 stride, valid padding, outputs 10x10x48
2: RELU	
2: Max pooling	2x2 stride, outputs 5x5x48
2: Flatten	
3: Fully connected	Input: 1200, output: 400
3: RELU	
3: Dropout	
4: Fully connected	Input: 400, output: 200
4: RELU	
4: Dropout	
5: Fully connected	Input: 200, output: 43

4. Hyperparameters

The code for training the model is located in the 7th and 9th cell of the Jupyter notebook.

I decided to train the model using the optimizer suggested in the LeNet example: AdamOptimizer. In the end I used 600 epochs and batch size of 1024.

The batch size was from how much memory the computer I was running on had available.

For the epochs number you need the network not to overfit or underfit. If the accuracy is still improving when you stop, it means you have probably underfit your network. Similarly if the network has stopped at a certain accuracy for a very long time, it probably means your network is overtrained, and has overfitted it's weight, and will most likely perform poorly on values it has not seen before. We need to strike a balance between underfit (still more to learn), and overfit (not general enough). Ideally this should be detected automatically and terminate the training when the accuracy has stabilized.

I wen for a manual approach, and tuned the epochs number manually and adjusting to a number that I found acceptable for the used learning rate.

For learning rate I tested a few options, but ended up using 0.001. You want a network that converges toward a solution fairly fast, but not too fast. Based on my thin experience, I decided I liked the converge rate of 0.001 better than higher or lower values.

Approach taken for finding a solution

The code for calculating the accuracy of the model is located in the 12th cell of the Jupyter notebook.

My final model results were:

- validation set accuracy of 97.6%
- test set accuracy of 96.1%

My approach contained many iterations. Here are the most important listed:

- At first I tried plain LeNet network without any normalization or regularization of any kind. That resulted in a poor performance of

around 80% accuracy.
- Next step was to normalize the data, but still not real improvements.
- As a last step I introduced dropout, and set the dropout rate to 0.9. This did not give too much improvements either.
- I extended the network to include more nodes at each level, and was able to increase the accuracy to about 95% validation accuracy, but only 91% testing accuracy.
- I decided to increase the number of epochs to 600 and also set the dropout to 0.5
- I tried in a couple of different ways to increase the number of nodes in the model, but neither more nodes in the convolutional layers, nor in the fully connected layers gave any better results. I reverted back to the delivered solution, which seems to be a good tradeoff between complexity and runtime.

Test a Model on New Images

Here are five German traffic signs that I found on the web:

My intuition was that image 3 and 5 could be difficult to classify due to the fact that they are skewed a bit and image 3 is also slightly rotated.

Model's predictions on these new traffic signs

The code for making predictions on my final model is located in the 14th, 15th and 16th cells of the Jupyter notebook.

Here are the results of the prediction:

Image	Prediction

Traffic signals	Traffic signals
Ahead only	Ahead only
Speed limit (70km/h)	Speed limit (20km/h)
Road narrows on the right	Road narrows on the right
Priority road	Priority road

The model was able to correctly guess 4 of the 5 traffic signs, which gives an accuracy of 80%. This is not too bad considering I have not trained the model with any kind of skewed images. My intuition of the difficult images was also confirmed, as image 3 turned out to be wrong. But for image 3 it was still classified as a speed limit sign. With only the digit 2 instead of 7.

3. How certain the model?

The code for making predictions on my final model is located in the 17th cell of the Jupyter notebook.

Image 1

The top five soft max probabilities were

Probability	Prediction
.99	Traffic signals
.000033	Beware of ice/snow
.00000071	Priority road
.000000689	Keep right
.000000088	Right-of-way at the next intersection

Image 2

The top five soft max probabilities were

Probability	Prediction
1.0	Ahead only
5.92588723e-24	Turn left ahead
7.52890552e-29	Go straight or right
8.07868461e-31	Turn right ahead
1.60519613e-33	Speed limit (60km/h)

Image 3

The top five soft max probabilities were

Probability	Prediction
.47062558	Speed limit (20km/h)
.09889314	End of all speed and passing limits
.09530148	Right-of-way at the next intersection
.07172929	Roundabout mandatory
.07053854	Bicycles crossing

Image 4

The top five soft max probabilities were

IMAGE4

Probability	Prediction
.48397925	Road narrows on the right
.43195409	General caution
.04340653	Traffic signals

.00889096	Road work
.00502721	Dangerous curve to the right

Image 5

For image 5 the network were extremely confident. Did not even consider other options. The last 4 options are simply the first 4 in the output.

The top five soft max probabilities were:

Probability	Prediction
1.0	Priority road
.0	Speed limit (20km/h)
.0	Speed limit (30km/h)
.0	Speed limit (50km/h)
.0	Speed limit (60km/h)