Traffic Sign Recognition

Alberto Rivera
March 24, 2017

Self Driving Car Nanodegree

Domain Background

This writeup and the accompanying jyputer notebook at part of the project 2 of the first term of the Udacity Self Driving Nanodegree program.

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

The template for this project can be found here:
https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project
My project also takes elements from a project done by Jeremy Shannon found here:
https://github.com/jeremy-shannon/CarND-Traffic-Sign-Classifier-Project/blob/master/Traffic_Sign_Classifier.ipynb

Rubric Points

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

Files Submitted

The files for this project include:
- CarND_P2_writeup.pdf (This write up)
- Traffic_Sign_Classifier.ipynb
- Traffic_Sign_Classifier.html
- The new-traffic-signs-data folder, which contains images not found in the GTSR dataset

Data Exploration

Dataset Summary

The code for this step is contained in the second code cell of the IPython 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 test set is 12630
- The shape of a traffic sign image is (32, 32, 3)
- The number of unique classes/labels in the data set is 43

Exploratory Visualization

The code for this step is contained in the third and fourth code cells of the IPython notebook.

Number of Dataset Images per Label

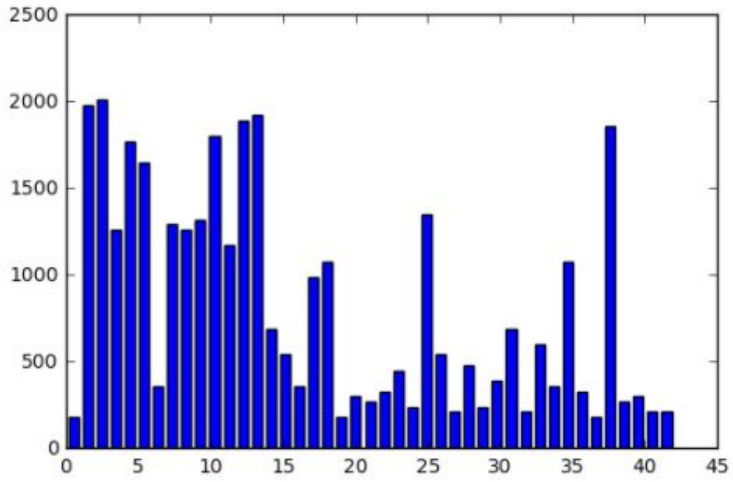

Figure 1

From Figure 1 we can see how the image types in the dataset are not uniform. This will be improved later in this project.

Selected Dataset Image Examples

Figure 2

Figure 2 shows an image corresponding to the first 14 labels. By consulting the signnames.csv file we can correlate the labels to the sign name/meaning. For example images with label “0” correspond to Speed limit (20km/h) images. More examples can be found in the output of cell 3 in the notebook.

Design and Test a Model Architecture

Preprocessing

The code for this step is contained in the fourth code cell of the IPython notebook.
As a first step, I decided to convert the images to grayscale. The reasoning behind this has more to do with the practicality of the particular implementation of this project rather than with achieving better results. The template for the implementation of this projects is Yann LeCun’s LeNet Model for identifying handwritten numbers. That model took in grayscale 32 x 32 images. Our model will do the same. Taking small grayscale images as input rather than larger color images is also much less computationally intensive and uses less resources, therefore saving time (and money if you are using AWS instances!).

Here is an example of a traffic sign image before and after grayscaling.

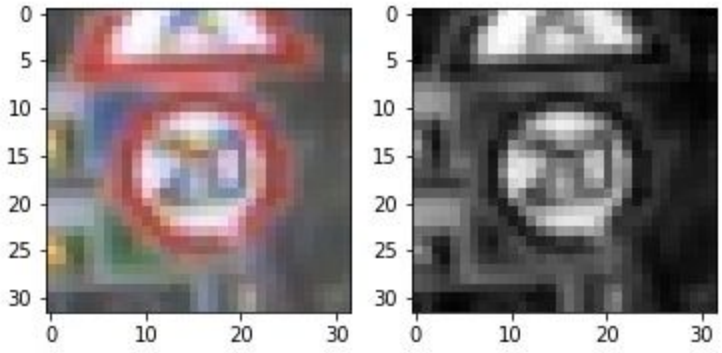

Figure 3

As a second step, I normalized the image data because having a wider distribution in the data would make it more difficult to train using a singular learning rate. Different features could encompass far different ranges and a single learning rate might make some weights diverge.

Lastly I create new images for those labels that have less than 400 examples. This is done by taking the existing images of such labels and making slightly different copies of them using various methods such as scaling, warping and adjusting the brightness. This is done to have a more uniform distribution of the data and increase the accuracy of the classification of those images that didn't have many corresponding examples in the dataset.

Model Architecture

The model used in this project in a modified version of the LeNet Model architecture by Yann LeCun. The modifications are also adapted from a research paper published by Pierre Sermanet and Yann LeCun.

ConvNet Architecture

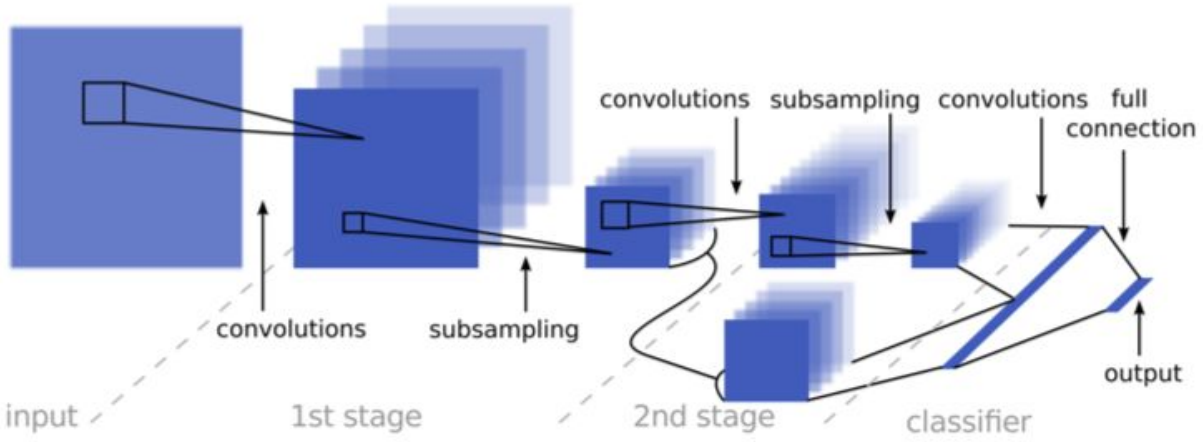

Figure 4

The implementation of the model is done seventh code cell of the jupyter notebook, in a function named LeNet. The model used in this project is an adaptation of Sermanet's and LeCun's model done by Jeremy Shannon. The model has the following setup:

1. 5x5 convolution (32x32x1 input, 28x28x6 output)
2. ReLU activation function
3. 2x2 max pool (28x28x6 in, 14x14x6 out)
4. 5x5 convolution (14x14x6 in, 10x10x16 out)
5. ReLU activation function
6. 2x2 max pool (10x10x16 in, 5x5x16 out)
7. 5x5 convolution (5x5x6 in, 1x1x400 out)
8. ReLu activation function
9. Flatten layers from numbers 8 (1x1x400 -> 400) and 6 (5x5x16 -> 400)
10. Concatenate flattened layers to a single size-800 layer
11. Dropout layer
12. Fully connected layer (800 in, 43 out)

Model Training

The code for splitting the data into training and validation sets is contained in the fifth code cell of the IPython notebook. To cross validate my model, I randomly split the training data into a training set and validation set. I did this by using the train_test_split function provided by scikit learn.

The code for training the model is located in the eleventh code cell of the ipython notebook. To train the model we feed the training section of the dataset to our training_operation. This section remains largely unchanged from the original LeNet code including batch size and the type of optimizer. Two considerable changes were the learning rate and the number of epochs. After running the experiment a few times I found a learning rate of 0.0009 to be most effective. The number of epochs was increased to 40 because many further epoch do not provide much improvement.

Solution Approach

The code for calculating the accuracy of the model is located in the eleventh cell of the Ipython notebook. To train the model we create a Tensor Flow session and run our training operation for a number of Epochs. I started with the unchanged LeNet model, and not augmenting the training data. Experimenting multiple times with different learning rates and epochs this first approach resulted in a validation accuracy of around 85%. While this was a great starting point since we only had to improve the result by less than 10% figuring out how to do this was quite challenging. In the end I chose to model my solution after Jeremy Shannon's who modeled his solution after the Sermanet and LeCun paper. Serment and LeCun achieved an accuracy of 99.17%. Shannon reached an validation accuracy of 99.3%. My best result to the date of writing is a validation accuracy of 98.4%, enough for the purpose of this project but far from the performance of the models mentioned.

My final model results were:

- Validation set accuracy of 98.4%
- Test set accuracy of 92.3%

Test a Model on New Images

Acquiring New Images

Figure 5

To acquire new german traffic sign images I used google street view around Munich and took 32x32 screen shots of some of the images I saw. Figure 5 shows the 6 images I found and used to test the model. I tried to take images that were similar to those already in the dataset. Having them be the same 32x32 shape as the dataset images saved us the extra step of re-scaling them.

The images were manually given labels by comparing them to the signnames.csv file and the correlated images on the dataset. The first image was assigned a label of 38, Keep right. This image should be straightforward to classify since it is very clear. However the actual sign is not centered in the image which might make it harder to classify. The second image has a label of 33, Turn right ahead. This one is also very clear and has high

contrast between the sign at the background. It is also very clear and mostly centered. The third image is labeled as 17, No entry. While the sign is very visible there is a lot going on in the background which has some foliage, shadows and a pole next to the sign. The fourth image is labeled as 11, Right-of-way at the next intersection. This image is very similar to other signs of the same shape with the main difference being the small shape in the middle of the sign, which can be hard to distinguish in such a small image. The fifth image has a label of 35, Ahead only. This image might be hard to classify because it has another sign partially visible on the lower right corner. However that partially visible sign does not correspond to one of our labels. The last sign has a label of 25, Road work. This image is in a similar situation as the fourth image, being only distinguishable by the small figure in the middle of the sign.

Performance on New Images

The classification accuracy of our model on the new images was 83.3%. This is because the model misclassified the one of the images. The accuracy is much lower than in our previous test of the model, but that could be because our new test set is very small.

Model Certainty - Softmax Probabilities

The code for making predictions on my final model is located in the fourteenth cell of the Ipython notebook.

Softmax Predictions

Figure 6

Interestingly when using softmax the model correctly predicts 100% of the images. However the predictions vary in certainty. The label of the first images is the lowest of

the first guesses with only 54% certainty, but it is still much higher than the second guess for the first image (39%). The second, third, and fourth images are correctly predicted with 100% certainty. The fifth image is also correctly predicted with a certainty of 93%. The second guess is visually a very similar sign so the slight uncertainty is understandable. The sixth image is correctly predicted with 63% certainty and the other guesses are images of the same shape.

Overall the performance of the model, while not as good as Shannon's or LeCun's , is still very impressive. I would like to revisit this project when I have a more complete understanding of convolutional neural networks and attempt even better scores.