Traffic Sign Recognition

This documents describes my approach to solve the Traffic Sign Recognition project according to the requirements. I've implemented a solution in python, not using the jupyter notebook template, in order to have more flexibility. During the development I've played around with different hyperparameters for the CNN as well as with the layout of the NN itself. Furthemrore I've made some investigations and adaptations on the test-data in order to realize a optimal learning input.
The following document is starting with a short overview of the goals for the project as it was given from Udacity. After that a short introduction to the architecture of the python solution is given. The following sections gives some details about the training set we've used for testing. With these findings, I'll provide some details about the preprocessing of the training data I've applied in order to get a better training set. After this sections I'll describe the model I've choosen for the project and give some comparisms of other testruns I did. The final section will summarize the performance on testsamples I've taken from my surrounding.

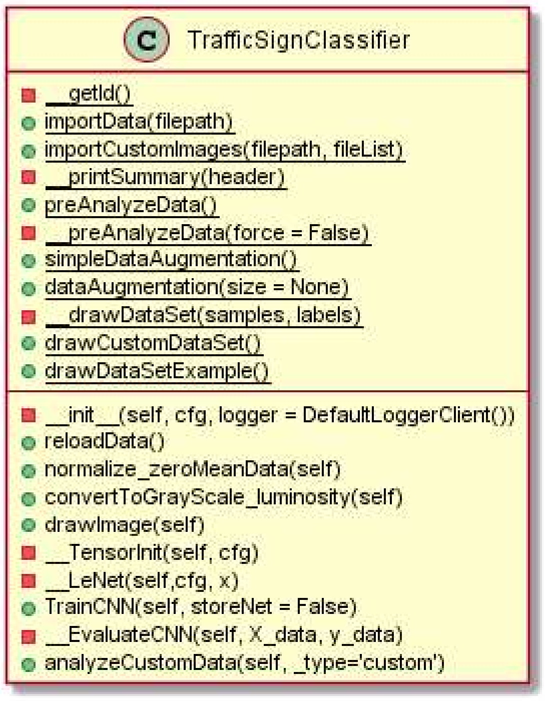
Build a Traffic Sign Recognition Project

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

TrafficSignClassifier.py overview

The first questions I've asked myself - how to deal with the degree of freedom that is given in a CNN like the LeNet architecture. I was expecting to make a bunch of measurements, changing some parameters and rerun the test. At the end a compareable result must be summarized. In order to make my life easier, I've decided to create a solution that is not based on the structure given in the jupyter notebook template. This solution allows me to predefine different settings of parameters and run the tests single/ multithreaded in a automatic sequence e.g. over night.


TrafficSignClassifier
__getId()
importData(filepath)
importCustomImages(filepath, fileList)
__printSummary(header)
preAnalyzeData()
__preAnalyzeData(force = False)
simpleDataAugmentation()
dataAugmentation(size = None)
__drawDataSet(samples, labels)
drawCustomDataSet()
drawDataSetExample()
__init__(self, cfg, logger = DefaultLoggerClient())
reloadData()
normalize_zeroMeanData(self)
convertToGrayScale_luminosity(self)
drawImage(self)
__TensorInit(self, cfg)
__LeNet(self,cfg, x)
TrainCNN(self, storeNet = False)
__EvaluateCNN(self, X_data, y_data)
analyzeCustomData(self, _type='custom')

The TrafficSignClassifier.py contains 3 classes (the TrafficSignClassifier and two helpers for logging) and a main function which is used to start the reading/training/verification process. I'll concentrate on the TrafficSignClassifier - further abbreviated as TSC - only - the LoggingClasses are used to write the results to console as well as in a textfile.

statics

TSC contains some static as well as instance members. The intention behind the static members and functions is, that reading training-data as well as applying different transformations on this data should be done once, and then shared among all the configurations.

As you can imaging: `X, Y train/valid/test/custom` contains the testdata read from the file. The `classes` member contains the association between label 0-42 and the corresponding textual representation. The class is storing an overview of the samples for each label in the `trainingLabelSetIndex` - meaning for each label I've the index to a training sample. The class member `id` is used in the multithreaded environment to associate log-outputs with the corresponding configuration setup.

Let review in short the static class methods:

importData(filepath):

Searches in the `filepath` for the files test.p/train.p/vlaid.p and signames.csv and uses `pickle` and `csv.DictReader` to read the data to the static classmembers mentioned above.

importCustomImages(filepath, fileList)

This method will import my custom data I've taken from my surrounding. The fileList is a list of sets containing (filename, labelNbr)- expecting *.bmp files. These informations will be stored in the static members `X_custom` and `y_custom` .

__printSummary():

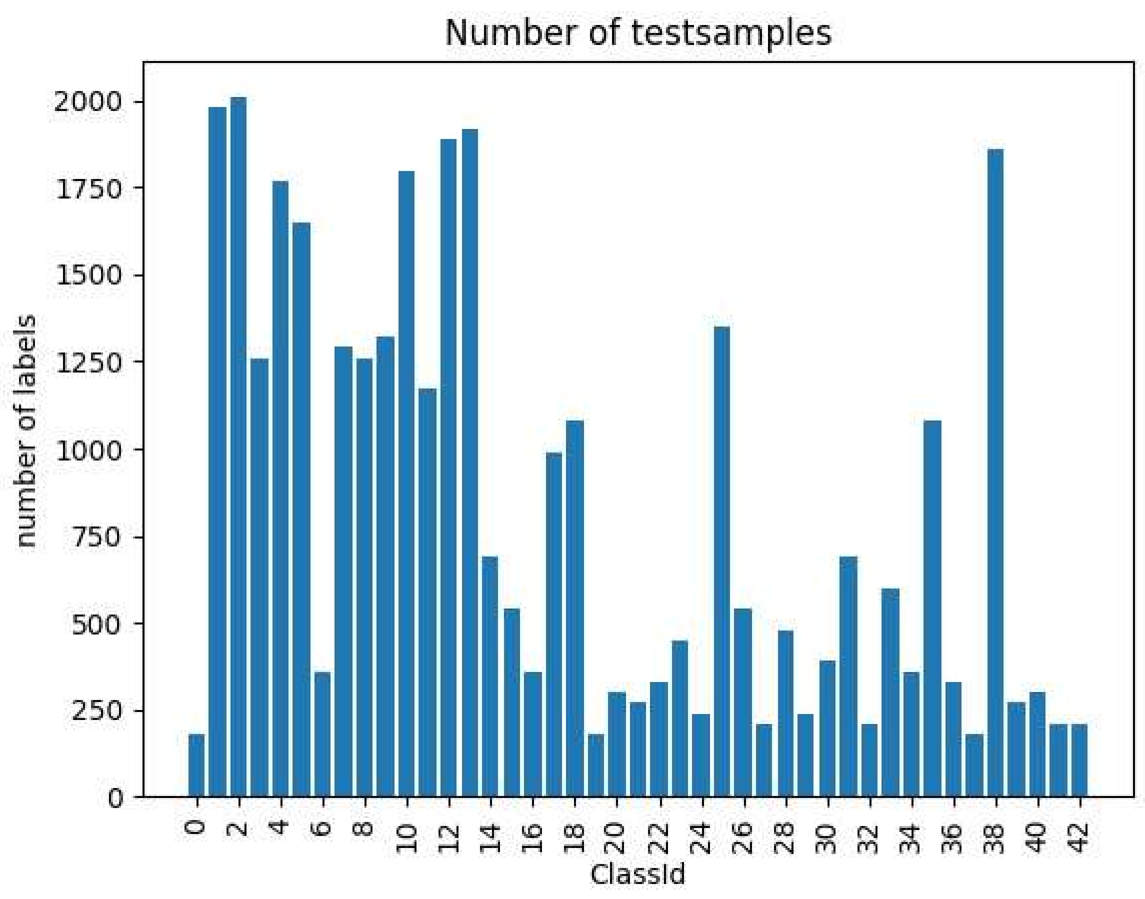
Provides a textual output based on the classmemebrs as expected by the jupyter notebook template in form of

> Basic Summary of the DataSet
> Number of training examples = 34799
> Number of validation examples = 4410
> Number of testing examples = 12630
> Image data shape = (32, 32)
> Number of classes = 43

Please note - this member is intented to be private and will be get called automatically e.g. in `importData`

preAnalyzeData()

This method provides a graphical representation of the samples for each label - the figure below shows you the exact distribution of label - samples.



__preAnalyzeData(force = False)

Again a private method which is getting called from other functions. In essence it uses the input data to fill the member `trainingLabelSetIndex` based on the given data. If the function was called once, it will immediately return in case that is is getting called another time. Only by forcing the function to reinitialize, the `trainingLabelSetIndex` will be updated. This method is used in order to visualize the augmented data set after aligning the samples for each label

simpleDataAugmentation()

The first try on augmentation was to simply adapt the number of samples for each label. This was done by identifying the label with the most samples and then cloning for each other label the samples so that at the end each label has the same number of test samples. During my testing I've figured out that beneath the drastically increased runtime, the performance of the CNN wasn't improving that good - so I decided to realize another function as described in the next section.

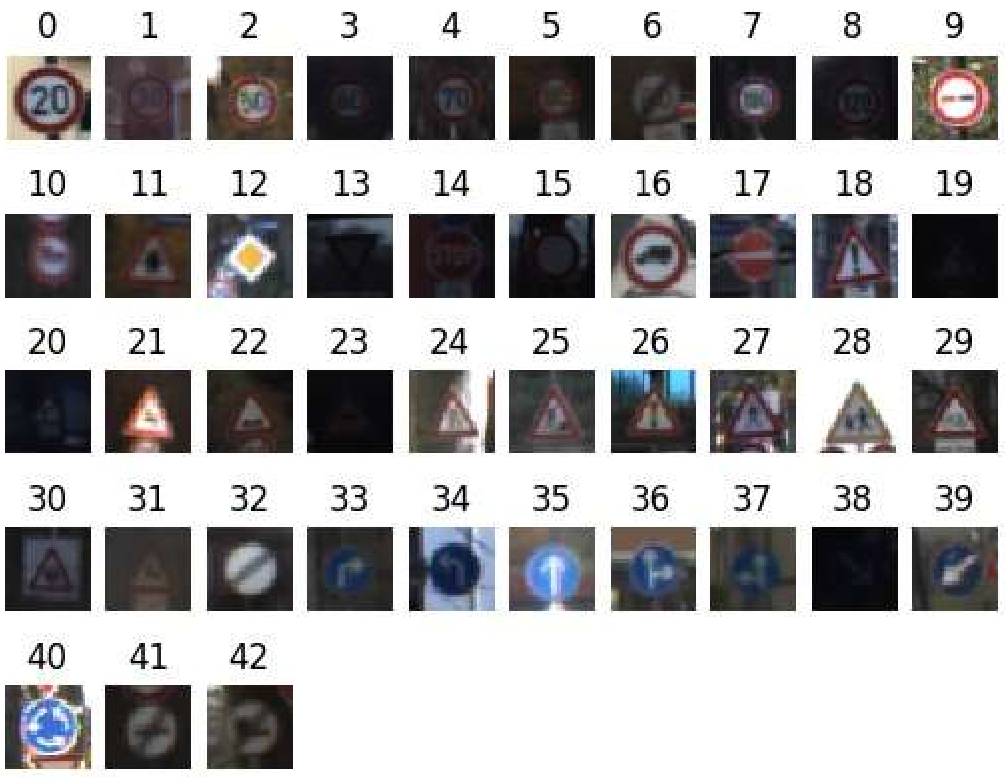
dataAugmentation(size = None)

As mentioned - the number of testsamples isn't equal for each label. Therefore I've realized a method which will change the samples in the following way. The paremeter `size` is used to specify the number of each sample for each label - `None` means to set the sample count to the max sample count of the given testdata - in our case 2010.
So what is happening in this function. For each lable I choose randomly `size` sample and then:

- use the first sample as it is
- use the second sample and rotate it for +15° without changing shape
- use the third sample and shift it along x,y axis for 2 pixel without chaning shape
- apply the operations on the next three samples

drawDataSetExample()

Take for each lable one sample and draw it via matplot.



instance members

A TSC instance is operating on the static members described above. The instance members reflecting configurations and operations as well as the layout of the underlying CNN network architecture. Each of the TSC instances is making a personal copy of the training/ validation/ test set so that any changes on the data is not interfering with other instances of the TSC running. I'll not go into the details of the member variables - these are more or less self-explaining. Beneath the testset the member variables represent the tensors used for training and validation.

Let discuss the member functions

normalize_zeroMeanData(self)

As a suggestion from the jupyter notebook template, i've implemented the zeroMean data normalization. This is done by changing each pixel using `X = (X-128) / 128` which will result in a mean of zero and a deviation of 1. I've noticed that this step improves the matching rate of the CNN immense - so in principle this is a standard operation i've applied to each testrun.

convertToGrayScale_luminosity(self)

Another suggestion from the template was to convert the images/ samples into grayscaled image. This is happening in this member function. For each pixel I'm applying `X_gray = X_red * 0.21 + X_green * 0.72 + X_blue * 0.07`.
The results/ effects of this transformations were negligible during my testruns - so I'm not applying this operation at all.

__LeNet(self,cfg, x)

As the name said, I'm staying with the LeNet architecture for my NN. The major difference to the LeNet architecture is the ability to configure via the `cfg` paraemter

- mu & sigma for the randomly created weights/ bias
- the scale of filters/ strides
- the scale of filters/ strides for max pooling
- the depth in the three fully connnected layers
- and an optional third convolution layer

Furthermore I've added a configureable **dropout** layer between FullyConnected1 and FullyConnected2 to prevent overfitting.

The remaining setup is more or less unchanged - meaning

- relu activation functions
- truncated_normal random functions for weights/ biases
- valid padding for filter-applying

And for sure - due to the fact that our resulting space contains 43 labels, the number of resulting logits is changed from 10 to 43.

Please note: this function is intended to be private - so it should only be called from other member functions, e.g. TrainCNN.

TrainCNN(self,cfg):

This is more or less the entry point for each TSC object. During TrainCNN a tensorFlow session is startet. The

sample-data is feeded via placeholders to the learning pipeline. As in the classical LeNet architecture our loss is measured as cross entropy of our results and the labels as OneHot encoded. This is translated into propabilities by using the softmax function. The resulting loss is given then by the reduced mean of these cross-entropy propabilities over the whole testsamples. The optimization - using AdamOptimizer - is running EPOCHS number of times.
As before - the `cfg` parameter is used to configure

- Batchsize which is by default 128
- number of epochs which is by default 10
- the learning rate which is by defaul 0.001

In case that our validation accuraty reaches the 0.93 mark after training, we immediately apply the NN to the training data. This is done via the `__EvaluateCNN` member.

__EvaluateCNN(self, cfg, X_data, y_data):

Evaluation of the trained NN using the test data. In contrast to training we've associated the dropout keep propability to 1.0. The accuracy is given by comparing the NN result for a sample with the label for the sample - which is either true (1) or false (0). The mean of this comparision over all samples provides the overall accuracy.

the configuration

The TSC is configured during startup by a python dict which contains different parameters to optimize. Some of these parameters are optional and will be evaluated by the TSC himself.

```
c_learningrate = "learningrate"   #default is 0.001
c_mu = "mu"                       # default is 0
c_sigma = "sigma"                 #default is 0.1
c_epoch = "epoch"                 #default is 10
c_batchsize = "batch"             #default is 128
c_keep_prop1 = "kp1"              #default is 0.5
c_keep_prop2 = "kp2"              #default is 0.5

{
  # filter shape + stride          maxpooling filter shape and stride
  "cv1" : ([5,5,3,32], [1,1,1,1]), "p1" : ([1,2,2,1], [1,2,2,1]),
  "cv2" : ([5,5,32,43], [1,1,1,1]), "p2" : ([1,2,2,1], [1,2,2,1]),
  # dimension of the hidden neurons - outputsize
  "fc1" : 120,
  "fc2" : 84,
  "labels" : 43,
  c_epoch : 15,
  c_learningrate : 0.001
}
```

The TSC expects to have at least 2 convolutions and three fully connected layers. The client can optionally add an additional covolution layer by specifying filter and stride as for the `cv1` and `cv2`.

```
"cv3" : ([2,2,43,43], [1,1,1,1]), "p3" : ([1,2,2,1], [1,1,1,1])
```

Optional parameters are the constants defined above like `c_keep_prop1` but also the depth of the fully connecteds. The resulting depths of the fully connnecteds is calculated on basis of the shape output of the convolution 2 or 3 and the expected number of labels (= 43) we're searching for.
I've applied a relation according to the LeNet architecture which results in

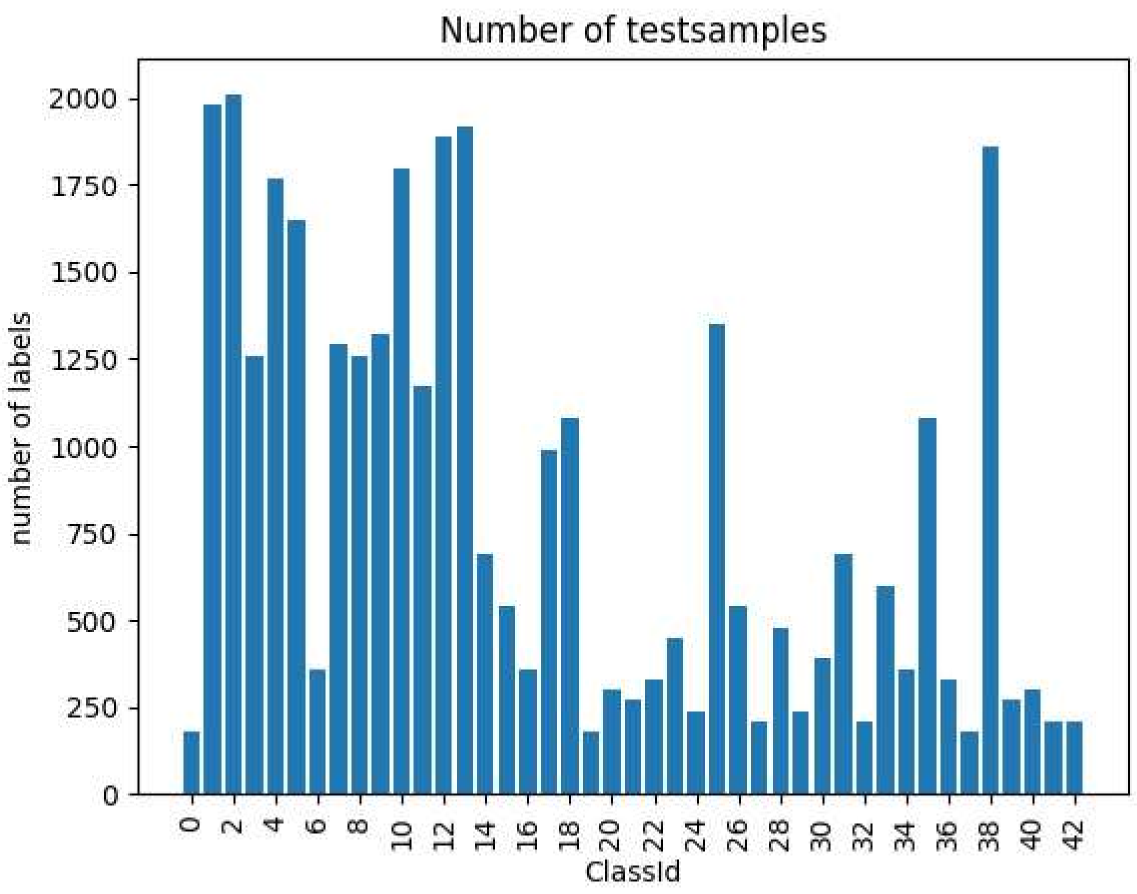
```
fc1 = (size(conv_out) + size(labels)) * 5 / 17
fc2 = fc1 * 7 / 10
#example LeNet architecture
fc1 = (400+10) * 5 / 17 = 120.58
fc2 = (120.58 * 7 / 10) = 84.40
```

Some configuration setup can be found in the solution

Data Set Summary & Exploration

The testdata available for us is partitioned into three segments. There is the training set which is used to train the NN. It has - please refere to the `__printSummary` method above - 34799 images of 32x32 pixels and a depth of 3 byte RGB. Each of the pictures represent one of 43 traffic signs as shown in `drawDataSetExample`.
The validation set contains about 4410 elements and is used to verify the accuracy of the trained network. Last but not least there is the test set with about 12630 samples. The test set is used to verify the trained NN on "new" data.

The first trainings of the NN shows a very low performance - so I decided to dive into the details of the testdata. As a first step I analyzed the distribution of samples for the different labels. As you can see in the figure below, there is a big variation of available samples for different labels.



E.g. label 2 has about 2010 training samples while lable 0 is below 250 samples. As a result, the resulting NN will be very good in classifying label 2 samples, but will often fail in case of labels 0, 19, 37, …

Furthermore while drawing different samples - I've picked out the label 41 - I realized that many of the images really look the same which would result in less samples.

Data preprocessing and augmentation

During the first testsetups I've worked with more or less the LeNet configuration to make some basic experiments. As the first step I've therefore concentrated on the training data and did not really play around that much with the tuning parameters of the NN itself. I'll go into more details about the NN architecture below - but now first of all concentrate on the data preprocessing.

To take care of the distribution of data samples and the similarity of samples, i've tried two approaches. First of all I concentrate on a even training of the NN overall labels. Therefore I started with a straight forward approach see `simpleDataAugmentation` method. The goal was to adjust the number of samples for each label to the max number of samples i've found - in other words - all labels should be trained with 2010 samples. Therefore I've created an index of all samples for all labels and appended randomly choosen samples of the corresponding label to the training set.
The result of this simply augmentation was moderate - training time and memory consumption increases, but the accuracy wasn't really much better.

So as a next step I've thought about how to add additional / new samples to the testset. The result is the `dataAugmentation` function. Similar to the `simpleDataAugmentation` it adds samples to the given set of samples. But instead of using the original samples, I've decided to let's say bring some noise in the testdata. This noise is realized by chose 3 samples randomly from the list of samples for a certain label:

- use the first sample as it is
- rotate the second sample for 15°
- shift the third sample for 2 px on each axis

At the end I've started my training with a sampleset of 1000 elements for each label. The training result was much better.

For sure - i've only applied two different operations to the sampleset. In order to increase the "noise" you could shifting/ rotating in negative direction or by more/ less degree/ px. I didn't spend more time on that, but I'm sure the robustness of the NN could be improved by providing more variance in the sampleset.

In addition to the augmentation, I spend some time in playing around with the samples itself. As we've learnt in out term, normalization of the inflow data is always a good start ;-). So I've implemented the `normalize_zeroMeanData` method. This method simply moves the average of each pixel to 0 and reduces the variance to 1. The consequence was meaningful and results in a better performance.
As the next step I've tried to play around with the colors - see `convertToGrayScale_luminosity` but the impact was disappointing so that is stopped further investigations.

So in short:

- augmentation
- normalization

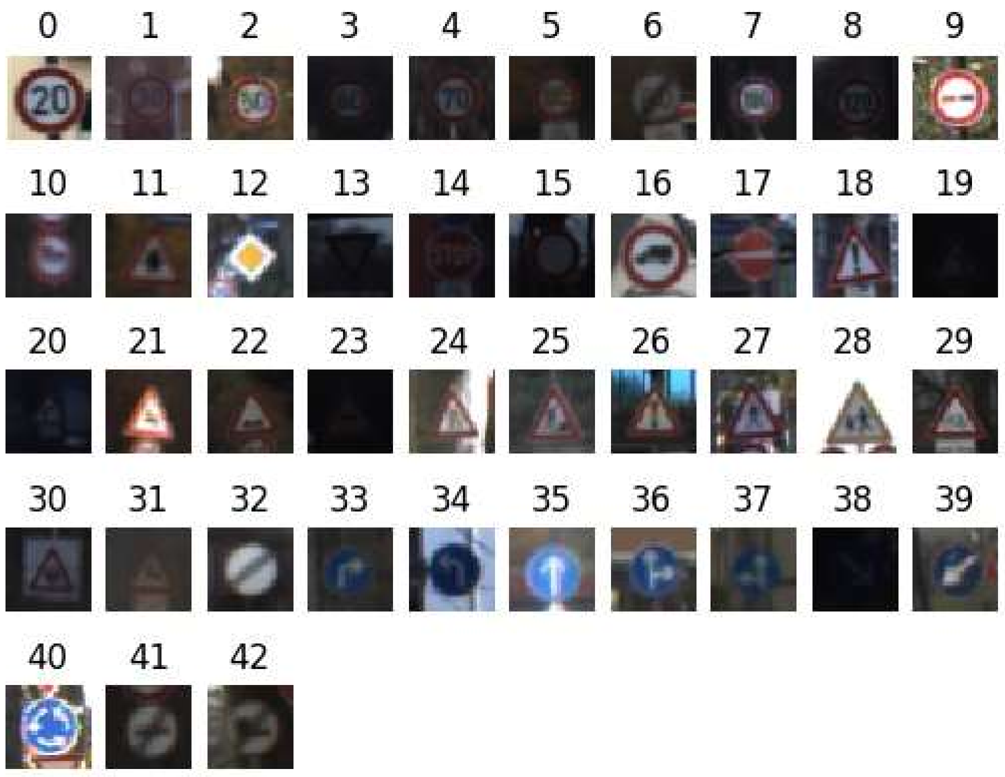
improved my NN.

1. Provide a basic summary of the data set. In the code, the analysis should be done using python, numpy and/or pandas methods rather than hardcoding results manually.

As the initial step in each training run, the TSC is reading the test/valid/train data that was offered to us. As mentioned earlier - even though there is a hugh amount of testdata - the testdata is not distributed very well among all labels.

> Basic Summary of the DataSet
> Number of training examples = 34799
> Number of validation examples = 4410
> Number of testing examples = 12630
> Image data shape = (32, 32)
> Number of classes = 43

Furthermore as you can see in the example image below, the quality - concerning brightness and resolution isn't really that great.



For example take label 13, 14, 15 - due to low brightness it very tough for a human to differentiate these images.

Design and Test a Model Architecture

As motivated in the section **TrafficSignClassifier.py overview** the first problem I wanted to address is to manage this high amount of degrees of freedom. During the development of the TrafficSignClassifier I already sheduled some testruns using the classical LeNet architecture - therefore I've decided to stay with this approach and enrich it by some additional measurements to reach better matching rate while lowering the overfitting risk.

One of the first steps I've made good experience with is the preprocessing of data I've described above. Augmentation of the data, aligning sample size and normalization to zero mean already increased fitting rate.

But as you can see below in the table, the classical LeNet architecture (Cfg7) wasn't proper designed to classify traffic signs. The major problem was, that the depth of the Convolutions were too small. The traffic signs offers much more features than the handwritten digits (concerning shape, color, edges…). So I decided very early to add another configurations which starts with a much higher depth like e.g. Cfg8 or Cfg1. With this adaptation I've already reached the goal.

Refering to the paper provided in the jupyter notebook, I increased the depth to a top of 128 - see e.g. Cfg3. But the result is compareable to configurations before (with smaller depths) so I dropped this approach.

As a next approach I've played around with the convultion depth itself - meaning adding another convolution layer. The question I was thinking about was - maybe two cnn aren't enought to merge abstract features to a more meaningful feature. So I adjusted the TrafficSignClassifier class in order to optionally add an another cnn layer. The situation I was facing very fast was that the output shape of the second cnn was already small - a third convolution on top will reduce it more. As a result - the input of the fully connected layers will decrease as well. My fear at this point was, by reducing the input to the fully connected will I loose the ability to recognize enough abstract features? As you can see in the testruns Cfg5, Cfg6 and Cfg12 - this was not the case. The accuracy of training was already above the requested 0.93 percent.

A major problem I was facing all the time was the problem, that the training accuracy was almost fine, but applying the resulting network on the testing data always results in a noticeable drop of the accuracy. I've reduced this problem by adding two dropout layers right behind fully connected 1 and 2.

So at the end, my testconfigurations I was working consists of:

- 2 to 3 convolutional layers
- 2 to 3 max-pool layers
- 3 fully connected layers
- 2 drop out layers between FC1 and FC2

Testruns

Please note: If not explicitly mentioned I've used for convolution filters always a stride of (1,1,1,1) while for the max pooling a stride of (1,2,2,1). The arrow in the table `<-` shall indicate the same setup as in the column to the left.

The following table represents trainings using the same input data but applying up to 6 different configurations. The following parameters are used for all of these configurations:

- learning rate = 0.001
- batchsize = 128
- epoch-times = 15

Parameter	Cfg1	Cfg2	Cfg3	Cfg4	Cfg5	Cfg6
Convolution1	5x5 3->32, stride(1,1,1,1)	4x4 3->16	5x5 3->108	5x5 3->43	4x4 3->43	4x4 3->108
MaxPooling1	2x2 , stride(1,2,2,1)	<-	<-	<-	<-	<-
ShapeOut CV1	14x14x32	14x14x16	14x14x108	14x14x43	14x14x43	14x14x108
Convolution2	5x5 32->43, stride(1,1,1,1)	5x5 16->43	5x5 108->108	5x5 43->108	4x4 43->108	4x4 108->43
MaxPooling2	2x2 , stride(1,2,2,1)	<-	<-	<-	<-	<-
ShapeOut CV2	5x5x43	5x5x43	5x5x108	5x5x108	5x5x108	5x5x43
Convolution3	-	-	-	-	3x3 108->43	2x2 43->43

MaxPooling3	-	-	-	-	2x2, stride(1,1,1,1)	<-
ShapeOut CV3	-	-	-	-	2x2x43	3x3x43
FullyConnected1	1075->120, dropOut = 0.5	<-	2700->120	2700->806	172->63	387->126
FullyConnected2	120->84, dropOut = 0.5	<-	<-	806->564	63->44	126->88
FullyConnected2	84->43	<-	<-	564->43	44->43	88->43
Accuracy Training	**0.960**	0.952	**0.959**	0.941	0.953	**0.958**
Accuracy Testing	**0.946**	**0.949**	0.929	0.933	0.938	0.939

One of the first suggestions when not reaching a good accuracy was to adjust the learning rate. So this table below applies almost the same configuration but changing the learning rate. Again the parameters for the testruns are:

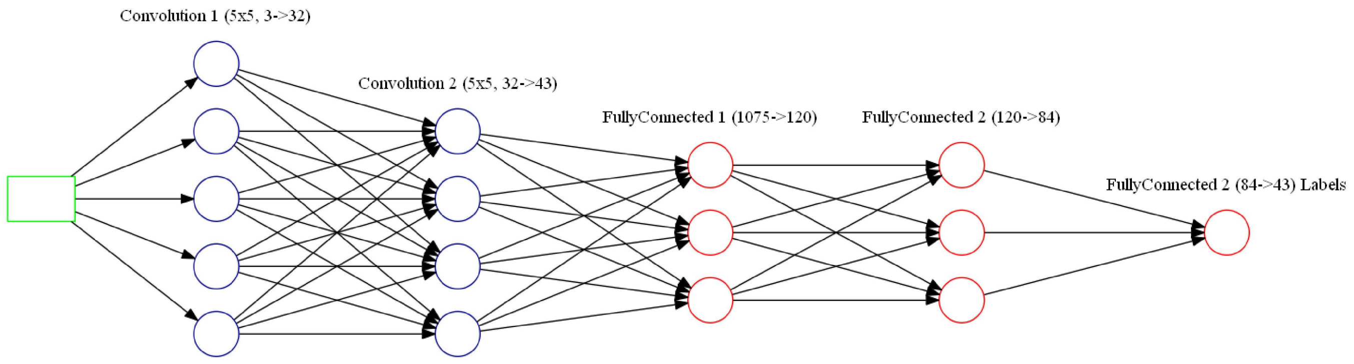
- learning rate = **0.0005**
- batchsize = 128
- epoch-times = 15

Parameter	Cfg7 (LeNet)	Cfg8	Cfg9	Cfg10	Cfg11	Cfg12
Convolution1	5x5 3->6, stride(1,1,1,1)	4x4 3->16	5x5 3->108	5x5 3->43	5x5 3->108	4x4 3->108
MaxPooling1	2x2 , stride(1,2,2,1)	<-	<-	<-	<-	<-
ShapeOut CV1	14x14x6	14x14x16	14x14x108	14x14x43	14x14x108	14x14x108
Convolution2	5x5 6->16, stride(1,1,1,1)	5x5 16->43	5x5 108->108	5x5 43->108	5x5 108->43	4x4 108->43
MaxPooling2	2x2 , stride(1,2,2,1)	<-	<-	<-	<-	<-
ShapeOut CV2	5x5x16	5x5x43	5x5x108	5x5x108	5x5x43	5x5x43
Convolution3	-	-	-	-	-	2x2 43->43
MaxPooling3	-	-	-	-	-	2x2, stride(1,1,1,1)
ShapeOut CV3	-	-	-	-	-	3x3x43

FullyConnected1	400->120, dropOut = 0.5	1075->120	2700->120	2700->806	1075->328	387->126
FullyConnected2	120->84, dropOut = 0.5	<-	<-	806->564	328->229	126->88
FullyConnected2	84->43	<-	<-	564->43	229->43	88->43
Accuracy Training	0.885	0.938	0.949	0.949	**0.959**	**0.957**
Accuracy Testing	-	0.924	0.938	0.930	0.936	0.937

Result and decision

As result of the different tests I've made, I decided to choose an architecture very close to the LeNet architecture - *Cfg1*.


Convolution 1 (5x5, 3->32)
Convolution 2 (5x5, 32->43)
FullyConnected 1 (1075->120)
FullyConnected 2 (120->84)
FullyConnected 2 (84->43) Labels

It consists of two convolutional layers with a filter of 5x5 3->32 and 5x5 32->43. Three fully connected layers are following these convolutions and result in a output of 43 labels. As in LeNet, the loss is the cross entropy of our results and the labels as OneHot encoded. I chose this architecture because it provides good results with a moderate consumption of resource - even though the resource consumption is negligible since it is only relevant for training.

A few words concerning number of epochs and learning rate: I measured in some cases that by increasing the number of epochs I was able to achieve better matching results - this didn't apply for the chosen configuration Cfg1 which was at its max accuracy right after epoch count of 10.
Furthermore I've changed the learning rate - that was one of the first suggestions learnt in the term. But as you can see, the impact is not noticeable.

The final result I've achieved with the jupyter notebook export is as following:

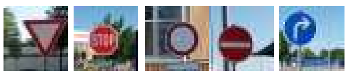
> EPOCH 1 …
> Instance 1: Validation Accuracy = 0.772
> EPOCH 2 …
> Instance 1: Validation Accuracy = 0.870

```
EPOCH 3 …
Instance 1: Validation Accuracy = 0.900
EPOCH 4 …
Instance 1: Validation Accuracy = 0.928
EPOCH 5 …
Instance 1: Validation Accuracy = 0.934
EPOCH 6 …
Instance 1: Validation Accuracy = 0.952
EPOCH 7 …
Instance 1: Validation Accuracy = 0.944
EPOCH 8 …
Instance 1: Validation Accuracy = 0.950
EPOCH 9 …
Instance 1: Validation Accuracy = 0.958
EPOCH 10 …
Instance 1: Validation Accuracy = 0.965
EPOCH 11 …
Instance 1: Validation Accuracy = 0.954
EPOCH 12 …
Instance 1: Validation Accuracy = 0.960
EPOCH 13 …
Instance 1: Validation Accuracy = 0.960
EPOCH 14 …
Instance 1: Validation Accuracy = 0.964
EPOCH 15 …
Instance 1: Validation Accuracy = 0.947
Instance 1: On testdata we’re achieving accuracy = 0.939
```

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.

I’ve taken some pictures in my sourrounding for this testsetup

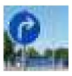


I’ve adjusted the pictures in the orientation (not all in the middle of the image).

2. Discuss the model’s predictions on these new traffic signs and compare the

results to predicting on the test set. At a minimum, discuss what the predictions were, the accuracy on these new predictions, and compare the accuracy to the accuracy on the test set (OPTIONAL: Discuss the results in more detail as described in the “Stand Out Suggestions” part of the rubric).

Here are the results of the prediction:

Image	Prediction
	Yield sign
	Stop sign
	No vehicles
	No entry
	**ERROR** Go straight or left instead of Turn right ahead

The model was able to correctly guess 4 of the 5 traffic signs, which gives an accuracy of 80%.

I was really suprised that the sample for label 33 was not classified correct. From the example pictures I’ve dumped (see provide a basic summary of the dataset) I’ve expected to run into problems with yield, stop or no vehicles. But the Turn right ahead was looking for me quiet clear.

3. Describe how certain the model is when predicting on each of the five new images by looking at the softmax probabilities for each prediction. Provide the top 5 softmax probabilities for each image along with the sign type of each probability. (OPTIONAL: as described in the “Stand Out Suggestions” part of the rubric, visualizations can also be provided such as bar charts)

For the yield sign the classifier wasn’t 100 sure, while for all other signs, except Turn right ahead, the classifier works with a reliable accuracy.
The suprising Turn right ahead which wasn’t event in the top 5 list. More crazy is the fact that the classifier provides a 99%+ rate for the “Go straight or left”.

Probability	Prediction
.66	Yield sign
.96	Stop sign

.99+	No vehicles
.99+	No entry
.00	Turn right ahead

The top 5 results of the customer images

Image	Top5 labels
	13 with 0.66, 14 with 0.34 12 with 0.00, 1 with 0.00 26 with 0.00
	14 with 0.96, 17 with 0.04 12 with 0.00, 29 with 0.00 10 with 0.00
	15 with 1.00, 13 with 0.00 9 with 0.00, 3 with 0.00 2 with 0.00
	17 with 1.00, 14 with 0.00 26 with 0.00, 18 with 0.00 30 with 0.00
	37 with 1.00, 39 with 0.00 40 with 0.00, 20 with 0.00 38 with 0.00

Summary

In this document I've provided an overview about my solution for the TrafficSignClassifier project. I've developed a class which allows me to setup different shapes of classifiers and run them in a batch like sequence. As basis for my classifier architecture I've reused the approach of LeNet.
The TrafficSignClassifier is applying different preprocessings on the training data (e.g. rotating images, normalize values) before it is used for training the network.
I've compared different network (difference in filter-shape, convolution depth, …) and decided to use more or less the LeNet configuration enriched by dropouts and adapted by higher depth in the filters of convolutions. As a result i've reached an accuracy of 0.95 on validation data and about 0.94 on test data.
Applying the classifier on my custom images results in a accuracy of 0.8 percent. Suprisingly one sign was completely mismatched.