Build a Traffic Sign Recognition Project

Sergiy Fefilatyev

The goals / steps of this project are the following:

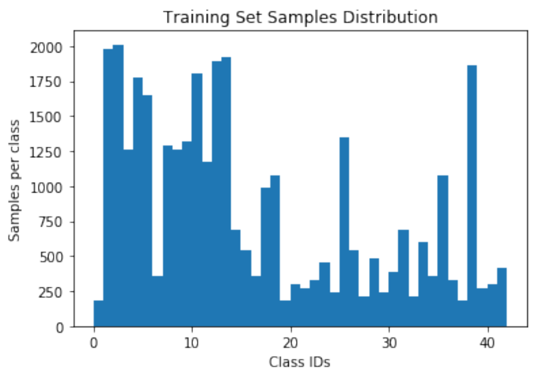
- •Load the data set (see below for links to the project data set)
- •Explore, summarize and visualize the data set
- •Design, train and test a model architecture
- •Use the model to make predictions on new images
- •Analyze the softmax probabilities of the new images
- •Summarize the results with a written report

Here I will consider the rubric points individually and describe how I addressed each point in my implementation. My project code is embedded within IPython notebook. Please take a look at "Traffic_Sign_Classifier.ipynb" within the same submission archive

Data Set Summary & Exploration

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

The code for this step is contained in the second code cell of the IPython notebook. I used the Matplotlib's "`hist()`" function to show the distribution of various classes within the training, validation, and testing sets. I used numpy's internal functions to calculate raw dataset sizes and the number of unique labels. The following represents short statistics of the dataset:

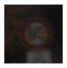
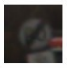


- •The size of training set is 34799
- •The size of test set is 12630
- •The size of validation set is 4410
- •The shape of a traffic sign image is 32x32x3
- •The number of unique classes/labels in the data set is 43

As it is visible from histograms (see IPython notebook), the training, validation, and testing sets draw samples from the same distribution (the shape of the histograms for those datasets is approximately the same).

2. Include an exploratory visualization of the dataset and identify where the code is in your code file.

The code for this step is contained in the third code cell of the IPython notebook. I used several modules in order to display images of the dataset in a nice HTML table. The table shows class ID, class name, sample image of a class, and number of examples.

Class ID	Name	Sample Image	Number of Examples
0	Speed limit (20km/h)		180
1	Speed limit (30km/h)		1980
2	Speed limit (50km/h)		2010
3	Speed limit (60km/h)		1260
4	Speed limit (70km/h)		1770
5	Speed limit (80km/h)		1650
6	End of speed limit (80km/h)		360

Design and Test a Model Architecture

1. Describe how, and identify where in your code, you preprocessed the image data. What techniques were chosen and why did you choose these techniques? Consider including images showing the output of each preprocessing technique. Pre-processing refers to techniques such as converting to grayscale, normalization, etc.

The code for this step is contained in the fourth code cell of the IPython notebook.

As a first step for pre-processing I did was normalizing the train, test, and validation data. I calculate the mean value and standard deviation for training data. Then for each train, test, and validation set, I subtract the mean for each example and divide it by the standard deviation. The normalization of data is important for training process as it helps the optimizer to converge faster.

I performed data augmentation using TensorFlow internal functions for that. This makes training extremely fast as augmentation is off-loaded to GPU. Having augmentation performed on-the-fly on CPU will make augmentation a bottleneck in the training process. In terms of augmentation techniques I used random brightness and contrast techniques. I am not showing augmented images as those are tricky to pick up from GPU :) Augmentation code is in cell 5.

2. Describe how, and identify where in your code, you set up training, validation and testing data. How much data was in each set? Explain what techniques were used to split the data into these sets.

I used default splitting of the data provided by the project (in the zip archive). See step 1 [cell 2 of notebook]. I made sure that the distribution of samples in training, validation, and test set look approximately the same. This is visible from the histograms - the shape of histograms for training, testing, and validation sets is approximately the same.

3. Describe, and identify where in your code, what your final model architecture looks like including model type, layers, layer sizes, connectivity, etc.) Consider including a diagram and/or table describing the final model.

The code for my final model is located in the fifth [5] cell of the IPython notebook. I used the default LeNet architecture provided in the class, but extended to three input channels. I also added a drop-out layer after fully connected layers to provide better regularization.

My final model consisted of the following layers:

Layer	Description
Input	32x32x3 RGB image
Convolution 5x5	1x1 stride, valid padding, outputs 28x28x6
RELU	
Max pooling	2x2 stride, valid padding, outputs 14x14x6
Convolution 5x5	1x1 stride, valid padding, outputs 10x10x16
RELU	
Max pooling	2x2 stride, valid padding, outputs 5x5x16
Fully connected	Outputs 120
RELU	
DropOut	Training probability of dropout 0.5
Fully connected	Outputs 84
RELU	
DropOut	Training probability of dropout 0.5
Fully connected	Outputs 43 logits
Softmax	Applies softmax for logits outputs

4. Describe how, and identify where in your code, you trained your model. The discussion can include the type of optimizer, the batch size, number of epochs and any hyperparameters such as learning rate.

The code for training the model is located in the sixth cell of the IPython notebook. To train the model I used an Adam optimizer. The batch size is 128 examples – inherited from LeNet's classroom exercise. The initial learning rate is 0.001, also inherited from LeNet's exercise. But after 15 epochs I drop it to 0.0002 (one fifth of the original) I trained the remaining 15 epochs with this learning rate. I found that dropping the learning rate gives about 1 percent accuracy boost on the validation set.

5. Describe the approach taken for finding a solution. Include in the discussion the results on the training, validation and test sets and where in the code these were calculated. Your approach may have been an iterative process, in which case, outline the steps you took to get to the final solution and why you chose those steps. Perhaps your solution involved an already well known implementation or architecture. In this case, discuss why you think the architecture is suitable for the current problem.

The code for calculating the accuracy of the model is located in the sixth cell of the IPython notebook. I used LeNet's architecture with the input of 3 channels as opposed to one in the original exercise. I also added a dropout layer to add regularization to the model. Instead of sub-sampling layer I used max-pooling and RELU instead of sigmoid activation (although these 'novelties' are just inherited from the classroom's exercise). The choice of LeNet is dictated by several factors: the model is small enough to be trained with little data the German Traffic Sign dataset provides. For larger models (VGG, ResNet, Inception) it would be necessary to have larger datasets or to used per-trained networks and then fine-tune it. Using per-trained networks is out of scope for this assignment.

Why do you think a convolutional layer well suited for this problem ?

Convolutional layers are well-suited for this problem because of several factors. Images as a class of input data provide redudancy in the sense that many neighboring pixels are correleated in the information they provide. Convlutional layers is the ideal instrument to capture this 'redundancy' and compress the number of parameters to model in the information within a layer. Fully connected layer would require hundreds times more parameters and would be problematic to train, especially with the dataset size of of this.

Why did you use pooling layers ?

I used maxpooling layer – a small change from the original LeNet architecture. Pooling layers reduce resolution, and, thus, compress the information provided to the next layer. It is a standart instrument of compositional architecture for neural networks where each consecutive layer is 'overseeing' a larger previous layer. Maxpooling allows further reduction of parameters in the network, preserving the most important features. MaxPooling also provides translational invariance for small shifts in image features since the output gets the maximum of the number of pixesl even if they shift somewhat.

Did you use dropout ? Why or why not ?

I did use dropout – another change from the standard LeNet architecture. It is one of the way to regularize the model, make sure the performance on the test set does not diverge much from the performance on the training set. For a small dataset like German Traffic Sign Dataset it is important to make sure the model does not overfit during multiple-epoch training. Drop-out is one of the available tools for this.

Can you justify your choice of activation ?

I used ReLu activation units as they provide much faster model convergence and are less suseptible to vanishing gradient problem. For this problem it was not absolutely necessary as the network architecture is not very deep. But it is still a better practice and allows moderate "modernization" of the old architecture. ReLu and their descendans have replaced sigmod functions which have problems for flow of gradients near the saturation levels of (0,1)

My final model results were:

- Training set accuracy of 99.5 %
- Validation set accuracy of 96.2%
- Test set accuracy of 94.1%

These results are quite satisfactory which proves the model is good enough for the problem.

There are plenty of possible improvements I could make. For example:

1. Use original resolution of images from the German Traffic Sign dataset. The receptive field of the classifier will increase proportionally, say to 224x224 pixels. All images are going to be resized into this resolution. Having higher resolution input images provides more information to the network. Also, images in higher resolution is easier to augment with various augmentation techniques (tilting, zooming-in-or-out, shearing, un-even darkening or lightening, perspective distortion, etc). I cannot understate how important augmentation is for a successful real-world application. A well-designed augmentation pipeline for training can improve various performance benchmarks (such as popular mAP) in tens of percents!

2. Higher resolution of input images will require an increase of the depth of the network with more advanced architectures (VGG, ResNets pre-trained on ImageNet and fine-tuned). The rule of the thumb is the deeper the network the more complex relationship it can model. The deep network usually have more parameters and, thus, require more data for good training. But this problem can be overcome by transfer learning, by populating early convolutional layers with parameters from baseline models trained on ImageNet and then fine-tuning last layers of the network with just images from the German Traffic Sign dataset. Model Zoo from Caffe's web-site is one of the sources for such pre-trained networks.

3. Have a classifier with multiple "heads", e.g. structured prediction. Instead of predicting just the class of a sign we could also predict the actual bounding box around the road sign within the crop. It is possible because the original German dataset provides not only classes for images but top-left and bottom-right corner of the bounding box around the sign. Such a classifier will be trained with multiple-task loss and will predict multiple task – class with confidence and bounding box corners.

4. Such a classifier from (3) will be easy to turn into a sign detector. First, in addition to the original 43 sign classes from the dataset we will need to introduce a "background" class – which outputs a "no sign" class when a background patch is shown to it. Then such a trained multi-task classifier is turned into a detector by "convolutionizing" the fully connected layers. Such a detector can be applied to various size images to both detect signs and localize their bounding boxes and predict their class.

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.

Here are ten German traffic signs that I found on the web (the actual images are located in the directory "test_set" inside the proejct):

Some images may be easily confused with other classes. For example SpeedLimit_30 looks very similar to SpeedLimit_80 as well as to other speed limits when scaled into 32x32. Speed limits may also be confused with other signs that have a sign's distinctive feature – red ring. One of such signs is "No Passing". The actual classifier provides 90% accuracy in my trained network. Only a single sign "End of SpeedLimit 80" was misclassified as "End of all speed and passing limits". But this is explainable because these two signs are quite similar in appearance, especially when scaled into 32x32. The top-2 label for this prediction is the correct label and it has a pretty high confidence = 20%.

2. Discuss the model's predictions on these new traffic signs and compare the results to predicting on the test set. Identify where in your code predictions were made. 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).

The code for making predictions on my final model is located in the eight's cell of the IPython notebook.

Here are the results of the prediction:

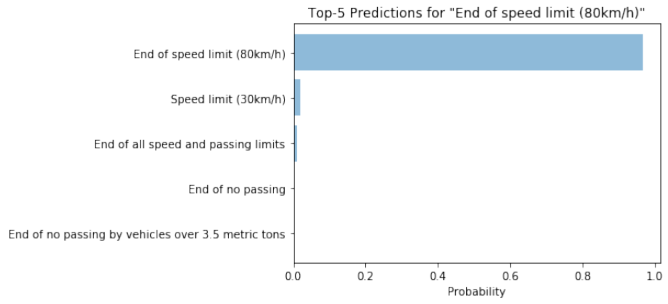
Image	Prediction
End of speed limit (80km/h)	End of speed limit (80km/h)
General caution	General caution
Keep left	Turn right ahead **(FAIL)**
Keep left	Keep left
No passing	No passing
Speed limit (30km/h)	Speed limit (30km/h)
Priority road	Priority road
Roundabout mandatory	Roundabout mandatory
Stop	Stop
Yield	Yield

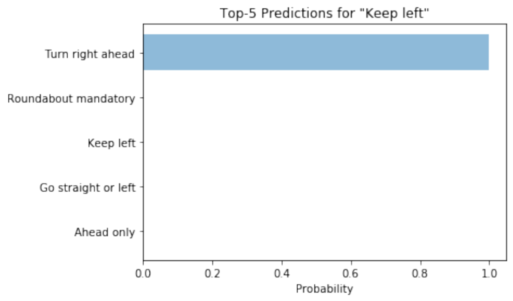
The model was able to correctly guess 9 of the 10 traffic signs, which gives an accuracy of 90%. This compares favorably to the accuracy on the test set of 12630 images.

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 and identify where in your code softmax probabilities were outputted. Provide the top-5 softmax probabilities for each image along with the sign type of each probability.

The code for making predictions on my final model is located in the 10th cell of the IPython notebook. I outputted the probabilities in terms of bar-charts from Matplotlib as suggested in rubric “Stand Out Suggestions”. Traffic_Sign_Classifier.html shows such top-5 predictions for all 10 test images from the web.

For the first image, “End of speed limit (80km/h)”, the model is very sure that this is the correct sign (probability of 0.96). The top five soft max probabilities were as in the left image below. For the third image “KeepLeft“ the network fails to predict the correct class. It confused “KeepLeft” with “TurnRightAhead”. It is understandable, as the pattern for the confused sign is very similar – blue circle with white ring and a white pattern in the middle. Fortunately, the correct “KeepLeft” prediction is within top-5 output.


Top-5 Predictions for "End of speed limit (80km/h)"
End of speed limit (80km/h)
Speed limit (30km/h)
End of all speed and passing limits
End of no passing
End of no passing by vehicles over 3.5 metric tons
Probability


Top-5 Predictions for "Keep left"
Turn right ahead
Roundabout mandatory
Keep left
Go straight or left
Ahead only
Probability