Machine Learning Engineer Nanodegree

Capstone Project: Traffic Sign Recognition with Convolutional Neural Networks

Antje Muntzinger
07th February 2019

I. Definition

Project Overview

Autonomous driving is one of the main research areas of artificial intelligence and machine learning. Traffic sign recognition has been available in advanced driver assistance systems since 2008 (https://en.wikipedia.org/wiki/Traffic-sign_recognition). First systems were developed jointly by Continental and Mobileye (see, e.g., https://www.mobileye.com/wp-content/uploads/2011/09/Continental-and-Mobileye-Joint-Press-Release-Combined-Technologies-from-Continental-and-Mobileye-Support-the-New-Speed-Limit-Information-of-the-New-BMW-7-Series.pdf)

Although research has been done for many years in this domain, there are still unsolved problems, such as computer vision in bad weather conditions, at nighttime, or additional traffic signs that are difficult to classify. I am particularly interested in this subject because I work at Mercedes-Benz and have written my PhD in the field of environment perception.

Problem Statement

In my capstone project I implemented a traffic sign detector. The detector gets images of traffic signs of different classes as input and returns the most likely class as output. The classifier should also correctly classify signs in different angles and lighting situations.

I used an image dataset with traffic signs where I first did some data augmentation and pre-processing, then I implemented a Convolutional Neural Network (CNN) for image classification. Images are fed into the neural net, the classifier is trained and later validated on a test set. CNNs are an adequate choice of neural nets for image classification because they conserve spacial structures in an image and are translation and rotation invariant, which is a desired property for a traffic sign detector.

Metrics

The performance of the model can be evaluated using accuracy score. This is appropriate since all benchmark models are evaluated using accuracy score as well. Accuracy is defined as

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{1}$$

I chose to use accuracy because for this application, it is most important to correctly classify a traffic sign. The probability of the class given by the neural net is only secondary. Therefore, log loss is not required (log loss would for example discriminate between a correct classification with high and low probability, respectively). Also, a false positive or false negative is not as critical as in other applications like, e.g., autonomous emergency braking, so no F-score is required, which would rate false positives higher than false negatives or vice versa.

II. Analysis

Data Exploration

In order to achieve this goal, I used the Belgian Traffic Sign Recognition Benchmark (https://btsd.ethz.ch/shareddata/). The training and testing sets can be found here: https://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Training.zip and https://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Testing.zip, respectively.

The Belgian Traffic Sign Recognition Benchmark is a multi-class, single-image classification challenge. It contains images from 62 classes in total and is a large, lifelike database. The images are taken in different angles and lighting conditions. Therefore, the use of this dataset is appropriate given the context of the problem.

The dataset consists of more than 7,000 images, where the training data comprises 4,575 images and the test data 2,520 images. The number of images per class varies from 6 images to over 300. The images also have different aspect ratios and sizes, they roughly vary around 128x128 pixels. Most images have a traffic sign nicely centered in the middle of the image, filling nearly all of the image. This makes the classification task easier.

I also calculated mean and standard deviation of the images: each of the three color channels has a mean of about 0.4 and a standard deviation of 0.2.

Exploratory Visualization

The visualization in Figure 1 shows a random image from each of the 62 classes. It can be seen that the images are already cropped in a way that the traffic sign is centered and fills most of the image. However, different aspect ratios and image sizes are visible. The different lighting and angles of the traffic signs can be seen as well. Also, it is clear that some classes are quite difficult to distinguish even for humans, e.g. class 5 and 6 or 15 and 16, respectively.

Algorithms and Techniques

To solve this classification problem, I want to use a Convolutional Neural Network (CNN). The CNN should get an image as input and give the probabilities of the classes as output. I will design a CNN from scratch in PyTorch rather than use transfer learning because I want to understand the neural net's behavior in detail.

The use of a CNN is appropriate for image classification because it conserves spacial structures of the image, unlike simple Multi-Layer Perceptrons (MLPs), for example. Clearly, neighboring pixels in an image are highly correlated, which should be taken account of in the classifier architecture.

Here is some more information on how a CNN works: It generally consists of a convolutional layer followed by a pooling layer. This can be repeated several times. Finally, there is a linear output layer.

Each convolutional layer calculates a convolution of the input image with a convolution kernel (a square matrix with odd size of lines and columns). The kernel moves over the image and the convolution is calculated for the current image area. The discrete convolution is defined as

$$I^*(x,y) = \sum_{i=1}^{n}\sum_{j=1}^{n} I(x-i+a, y-j+a)k(i,j). \tag{2}$$

Here $I^*(x,y)$ is the resulting pixel, I is the input image, a is the middle pixel in the square convolution kernel, and $k(i,j)$ is an element of the convolution kernel of size nxn. This method results

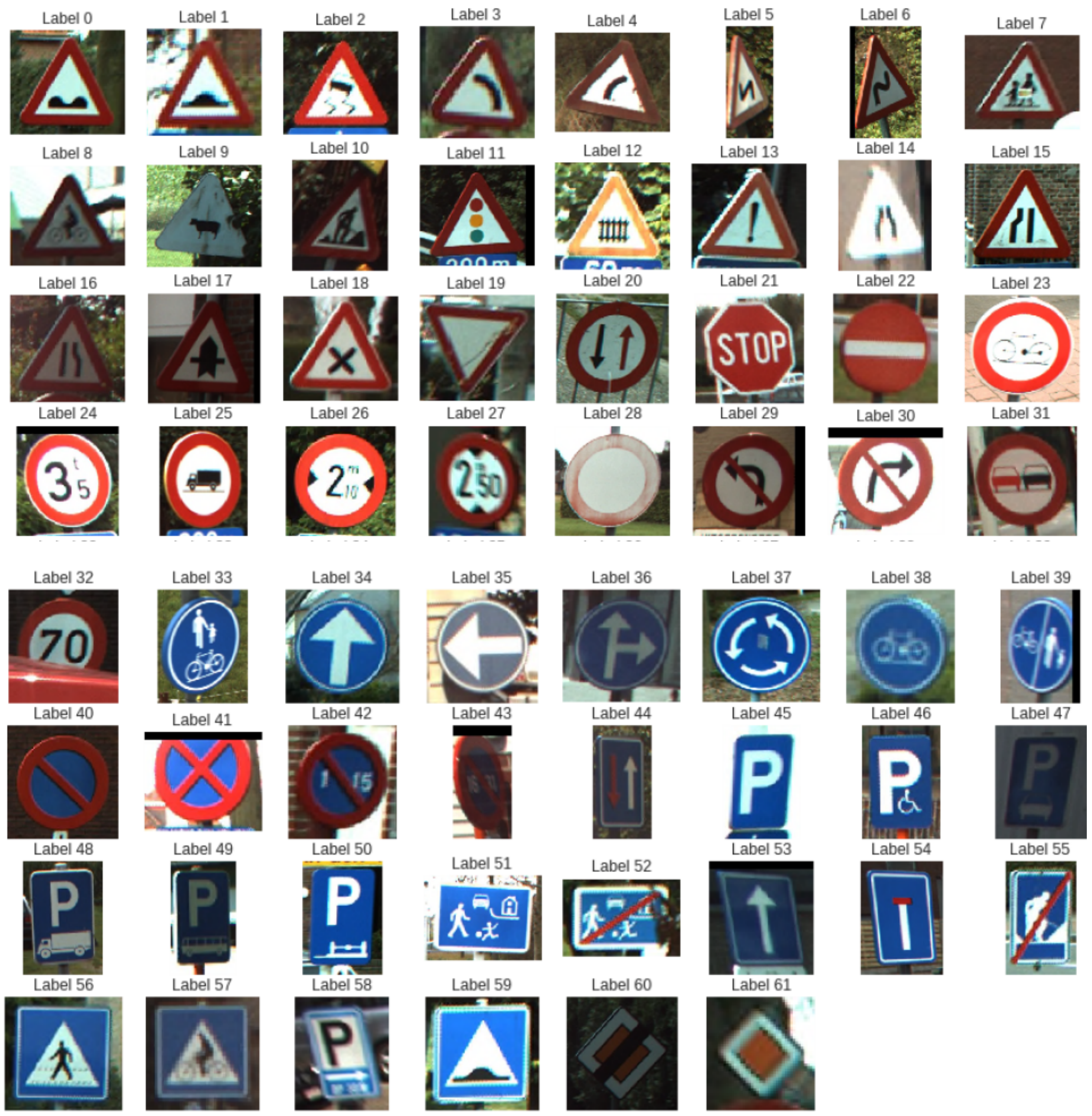


Figure 1: Visualization of a random image from each of the 62 traffic sign classes.

in an advantage of CNNs: They are translation and rotation invariant, which is very useful for traffic sign recognition.

A pooling layer reduces the size of the image. For example, a max pooling of kernel size 2 would reduce each 2x2 square of pixels to a single pixel containing the maximum value of the previous 4 pixels.

Finally, an activation function calculates the activation of the output of a layer. Often, a ReLU (Rectified Linear Unit) function is used, which is defined as

$$f(x) = \max(0, x). \tag{3}$$

The activation function models the activation of a neuron in biology, which only transmits a stimulus if a certain threshold is exceeded.

Benchmark

The paper "Traffic Sign Recognition – How far are we from the solution?" by Markus Mathias, Radu Timofte, Rodrigo Benenson, and Luc Van Gool can serve as a benchmark for this problem. It was published at the International Joint Conference on Neural Networks (IJCNN 2013), Dallas, USA, and can be found here: https://btsd.ethz.ch/shareddata/publications/Mathias-IJCNN-2013.pdf. Authors report that their models reached an accuracy between 95% and 99% without including traffic sign specific knowledge in the classifiers.

III. Methodology

Data Preprocessing

As mentioned above, the training data consists of 4,575 images and the test data of 2,520 images. I split the training data further into 80% training and 20% validation data (3,660 and 915 images, respectively). As the images also have different aspect ratios and sizes, I resized all images to 224x224 pixels, because this is the standard input size for transfer learning with PyTorch models. (I didn't use transfer learning, but wanted to keep the option for later comparison).

I then normalized the images to have a mean of 0 and a standard deviation of 1. This should make the classification problem simpler.

I also used image augmentation techniques to increase the number of training data: I randomly rotated the images and cropped them in different manners. This should make the classification more stable. I didn't flip them because obviously, the orientation is characteristic for some traffic signs. This can also be seen in Image 1 above: For example, Label 3 and Label 4 could not be distinguished if the images were flipped.

Finally, I loaded the training data as PyTorch tensors in batches of 16 images, respectively.

Implementation

I designed a very simple CNN consisting of two convolutional layers followed by two max pooling layers. The convolutional layers transform the three input color channels to 16 and 32 channels, respectively. The kernel size is 3, so each pixel value is convoluted with the eight surrounding pixel values (left, right, up and down neighboring pixels). I used a padding of 1, so the border pixels of the image can also be convoluted and the image size is not affected by the border pixels. I used a stride of 2, which means the convolutional kernel moves for two pixels after each convolution. Thus, image sizes are half as big after each convolutional layer.

Each convolutional layer is followed by a max pooling layer. I used a kernel size of 2 and a stride of 2, therefore each pooling layer also results in image sizes half as big as before. All in all, input images of size 244x244 pixels with 3 color channels are transformed to images of size 14x14 with 32 channels. Image width and length are transformed to much smaller sizes, whereas there are more channels that abstract features of the image.

The CNN finally has a linear layer. Therefore, the images of size 14x14x32 are flattened into a single vector of length 14*14*32 = 6272. The linear layer outputs 62 values, one for each class.

Both max pooling layers are followed by a ReLU activation function, which returns 0 if the input is negative, and returns the input for positive input. This activation function is very common because it has a simple derivative of 0 and 1 for negative and positive input, respectively, and thus doesn't suffer from vanishing gradients. I used a LogSoftmax as final activation function. Basically a Softmax function gives a value between 0 and 1 for each of the 62 channel outputs, thus giving a probability of the input sign to belong to each of the 62 classes. LogSoftmax gives the logarithm of these probabilities, which is computationally more stable: probabilities often lie close to 0 or 1, which can result in numerical problems. Taking the logarithm gives ranges between minus infinity and 0. Thus, to get the final class probabilities, the logarithm has to be reverted by an exponential function.

I implemented the model architecture using PyTorch. This is convenient because the library already contains powerful tools for solving classification problems with neural nets. I used an Adam optimizer with second order derivatives, which is also already implemented in PyTorch.

I trained my classifier on Google Colab for GPU support.

One thing I had to pay special attention to was the number of input channels from the last pooling layer to the linear layer. To calculate the correct number, I implemented a small function that takes as input the original image length and width, kernel size, padding and stride and returns the output size of the convoluted image given my model architecture.

I found it quite difficult to deal with all the different data types such as PyTorch tensors, cuda tensors of type double and float, NumPy arrays, etc. Many conversions were needed. Also, PyTorch expects the color channel to be the first dimension, but matplotlib assumes it is the third dimension, so several conversions were needed here as well.

Refinement

I first used a random flipping of the input images as image augmentation. However, I noticed that this doesn't make sense and the classifier cannot distinguish between some classes this way, as I explained above. Therefore I removed this transformation.

I started with a learning rate of 0.0001 for 100 epochs, but noticed that the best accuracy remained constant over the last approximately 50 epochs. The accuracy was below 80%. Therefore, I restarted the training with a decreased learning rate of 0.00005 for 20 epochs, and finally with a learning rate of 0.00001 for another 20 epochs. This gave me the final accuracy of about 83%.

IV. Results

Model Evaluation and Validation

During training, performance of the classifier was evaluated after each 100 batches of training data. The current accuracy was calculated on the validation set, and the model was saved if the accuracy had improved. Also, training and validation losses were monitored to avoid increasing validation losses, which could be a sign for overfitting.

After training, the final model was evaluated using images from the test set. The test set had never been used before (performance of the classifier was only monitored on the validation set), therefore this analysis can be trusted and the classification can generalize to new datasets. I first plotted some random test images and their predicted classes. Most of the predictions were true; however, the highest probability was not always close to 1. Some classes were correctly predicted with a probability of only about 0.5. These images were quite blurred or badly cropped and sometimes even difficult to classify for humans. All in all, the classifier performs well with unseen data and its results can be mostly trusted.

I started with a learning rate of 0.0001 and decreased it gradually to 0.00001. This makes sense because first the classification can be improved a lot quite fast (with a higher learning rate), but later fine tuning has to be done, which requires a lower learning rate. Also, the classification is quite robust because of the data augmentation techniques, random resizing and cropping helps the classifier generalize to new data.

Finally, the overall accuracy of the classifier was evaluated on the whole test set. The analysis was accurate with about 83%.

Justification

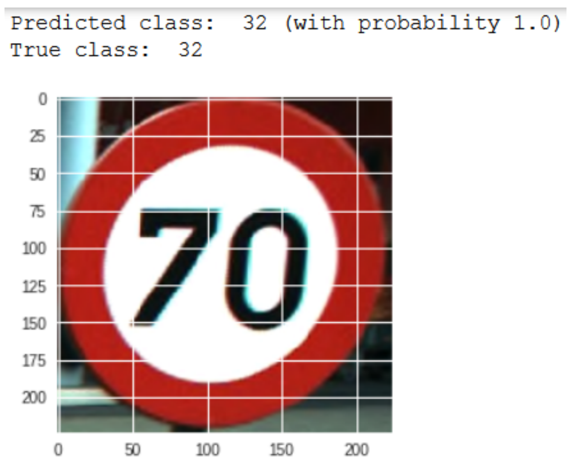
My traffic sign detector classified signs of the test set with an accuracy of 83%. The benchmark publication reports an accuracy between 95% and 99% (https://btsd.ethz.ch/shareddata/publications/Mathias-IJCNN-2013.pdf). Therefore, there is still room for improvement. However, I used a very small CNN because I wanted to understand the net's structure and setup rather than get a score as high as possible. The performance could of course be improved using transfer learning instead of building a neural net from scratch. Also, I used a simple architecture with only two convolutional layers, two pooling layers and a linear layer. Using more layers could improve the results as well as using dropout to avoid overfitting.

In a real world application such as autonomous driving, the classifier would be trained with much more computational power, thus resulting in a besser overall performance. In addition, an autonomous vehicle doesn't use the raw classification results of each single image, but applies some tracking algorithms on the detections. For example, a Kalman filter could be used with the image detections as input. The result would be a more stable and robust object detection filtered over consecutive images. Therefore, single wrong classifications or missing detections don't matter in practice.

V. Conclusion

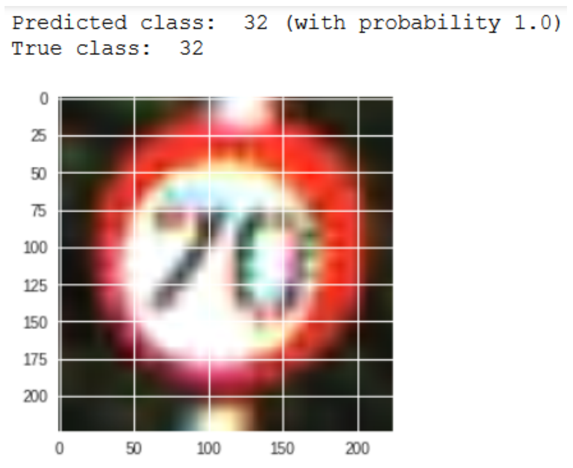
Free-Form Visualization

Figure 2 shows some random test images with different properties. Figure 2 (a), (b) and (c) show images that are quite simple to classify, they are very sharp and well illuminated. Therefore, the correct class is predicted with a very high probability of 1.0 or 0.9. The images in Figure 2 (d), (e) and (f) are much more difficult to classify. The lighting is darker, the images are blurry or the cropping is not adequate. Therefore, although the image class is predicted correctly, probabilities are only 0.7 or 0.6. The last example image is even very difficult to classify for humans.



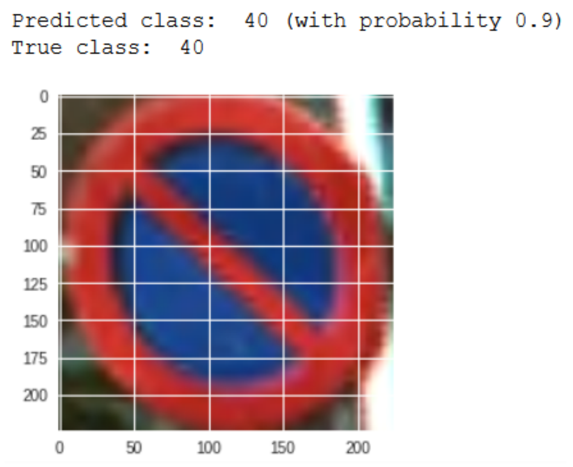
```
Predicted class:  32 (with probability 1.0)
True class:  32
```

(a)



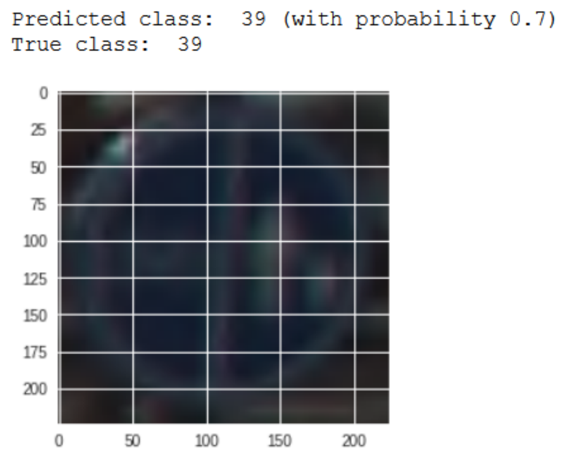
```
Predicted class:  32 (with probability 1.0)
True class:  32
```

(b)



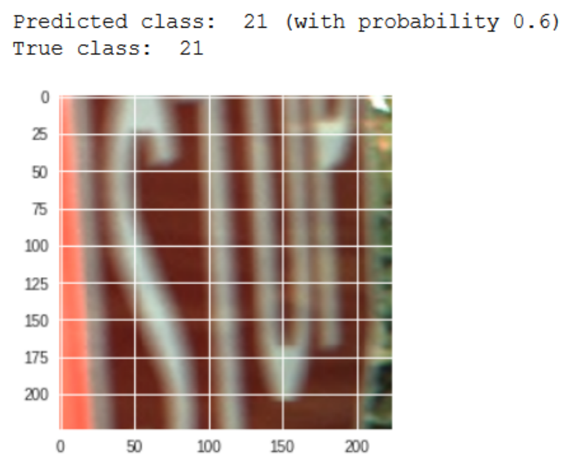
```
Predicted class:  40 (with probability 0.9)
True class:  40
```

(c)



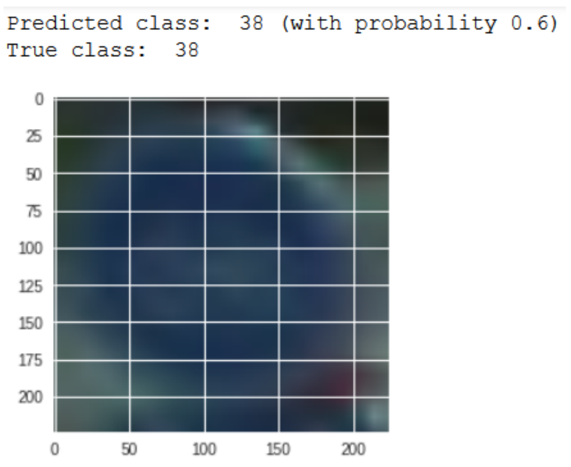
```
Predicted class:  39 (with probability 0.7)
True class:  39
```

(d)



```
Predicted class:  21 (with probability 0.6)
True class:  21
```

(e)



```
Predicted class:  38 (with probability 0.6)
True class:  38
```

(f)

Figure 2: Examples of different classification accuracies.

Reflection

All in all, I used the following steps to solve the traffic sign detection problem: I chose the appropriate dataset, did some basic data exploration and modifications, applied image augmentation techniques and loaded the data in batches. Then I defined my CNN and trained the classifier. Finally, I evaluated the overall performance on a test set. It was difficult to obtain an accuracy as high as in the benchmark publication. However, this was what I expected because I used a very simple neural net, as stated above. In a real-world application, e.g. in an autonomous vehicle, the classifier would of course have to be further improved.

Improvement

There are many ways to improve the performance. First, transfer learning would improve the classification results a lot because there are very powerful models available already. More layers could improve the results as well. Using dropout could avoid overfitting and make the classification more stable. More training data should also be used because the dataset used here is quite small. Finally, one could optimize hyper parameters further, such as learning rate, number of epochs, image sizes etc.