AWS Machine Learning Engineer Nanodegree Capstone Project: Recognizing Traffic Signs

Tom Schwarzburg

April 16, 2023

I. Definition

Project Overview

In the field of autonomous driving, computer vision plays a huge role. We need to extract data from the cameras, transform it, analyse the data and base our decisions on the input. Due to the high complexity of these problems, it is not feasible to generate these algorithms by hand. Rather, these often make use of convolutional neural networks and solutions like them.

This project will provide a solution for the German Traffic Sign Recognition Benchmark (*SSSI11*) and also an expanded problem. For a given photo of a traffic sign, we will decide in what group of traffic signs it falls and further also decide exactly what traffic sign the photo shows.

The dataset that I used is the one from the same Benchmark. As it is currently not available on the official site, it can also be downloaded from Kaggle.

Problem Statement

When we are driving a car, we have to make sure to collect and understand all the information from the outside of the car. The current weather situation or the visibility of possible pedistrians influence our driving decisions. One other important aspect is traffic signs. They tell us how fast to drive, warn us of dangers and overall influence our choices.

Autonomous cars also have to solve this problem. This is why the German Traffic Sign Recognition Benchmark focuses on actually recognizing them. With this information, we can then for example decide, how fast our car should drive.

Recognizing traffic signs is especially made difficult due to the high similarity between different signs. Speed limit signs often just differ in the number that is written on them. We're also driving in a fast moving vehicle, so we do not see them for a huge period of time and we might not get them at a perfect angle. Therefore the problem to solve is not trivial.

This problem is a multi-class classification problem. To solve it, I built two classifiers. Both are trained on the given dataset. One of them returns a specific group of the recognized traffic sign, which are taken from the Benchmark, while the other returns the exact traffic sign. The classifiers are based on convolutional neural networks, as they perform better than traditional computer vision algorithms. I first trained a default CNN with a pre-trained ResNet50 backbone on the data and then improved the results by applying hyperparameter optimizations.

Metrics

The chosen metric is based on the one used in the Benchmark. It uses Accuracy to compare different models, which is why it was chosen for my use case as well. It is calculated as follows:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives} \tag{1}$$

As we have a multi-class classification problem, we use multiple versions of accuracy to measure the performance of our models. We first use the overall accuracy of the model for all classes to rank our models. It will show how good the model is for everything.

To also find out which classes cause the overall accuracy to drop, I will also use the accuracy for each class as a performance metric. This will show which classes perform worse and will help to analyze the problems.

II. Analysis

Data Exploration

The dataset I used is the one from the German Traffic Sign Recognition Benchmark. Included in the data are different images of traffic signs and a csv with the ground truth. The images themselves are already each in a subfolder corresponding to the matching label.

We have around 50000 images of traffic signs with 43 different classes. The image size varies from 15 x 15 to 222 x 193 pixels. We will have to resize the images to the same size to later be able to use them.

As the ground truth is split into the 43 different classes, we will have to combine these classes ourselves to make it comparable to the benchmark version.

The producers of the dataset made sure that it only contains classes which occur with a certain frequency. It is for this reason that we should not have huge problems with underrepresented traffic signs.

Exploratory Visualization

The following figures represent the distribution of the different classes in the training, validation and testing datasets for each approaches, once for combined classes and once for distinct ones. The entries are ordered by the number of occurences descending.

We can see that as previously mentioned the distribution of the classes in each datasets is very similar, which means that the results for one dataset is very applicable to the other ones.

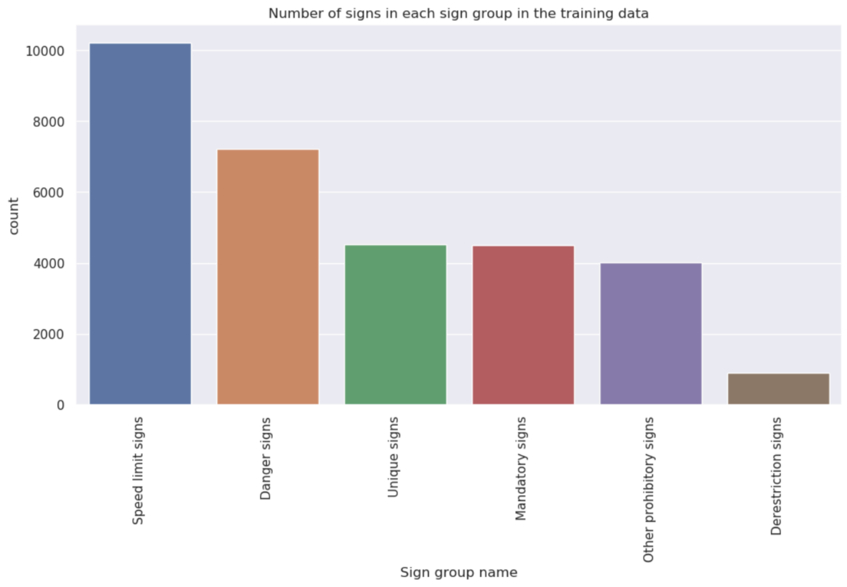

Figure 1: Number of occurrences of sign groups in training data

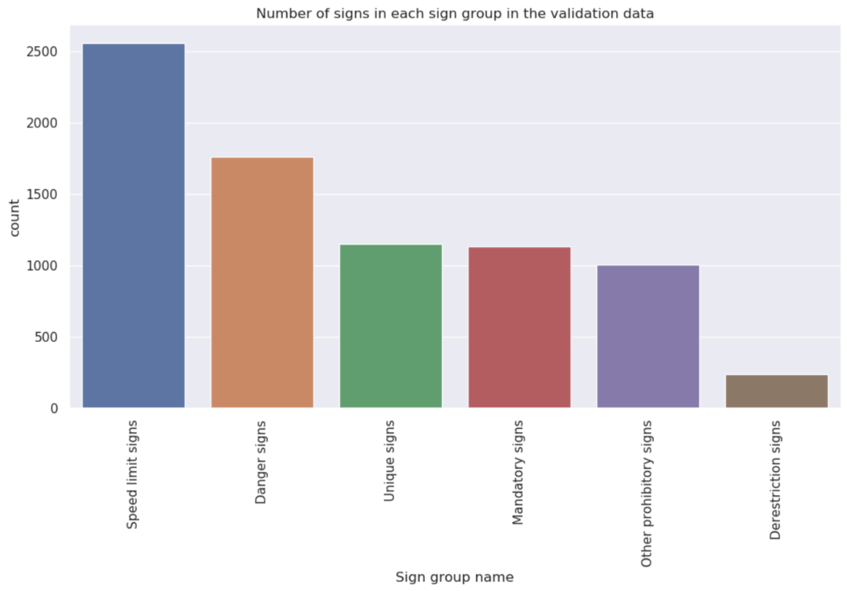

Figure 2: Number of occurrences of sign groups in validation data

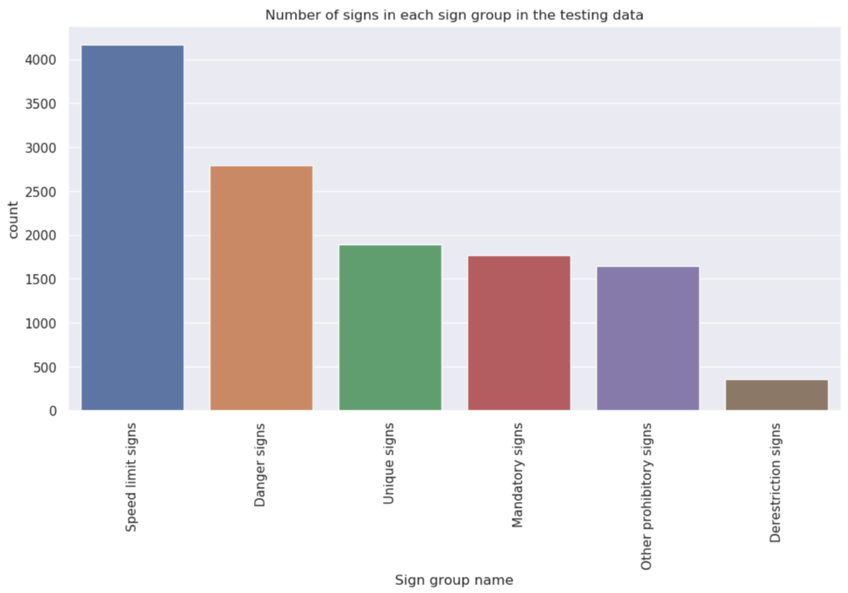

Figure 3: Number of occurrences of sign groups in testing data

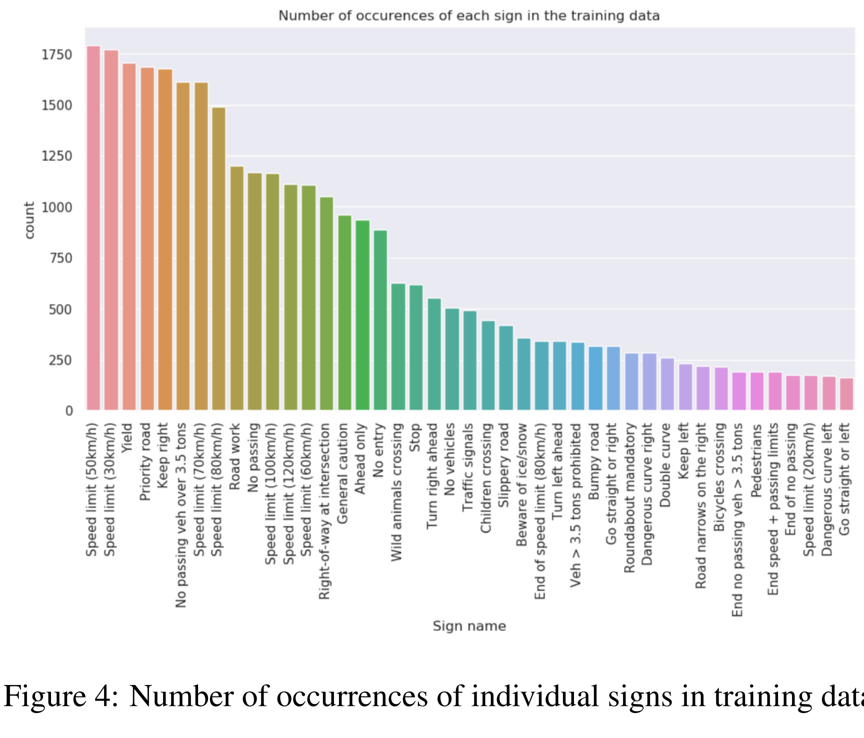

Figure 4: Number of occurrences of individual signs in training data

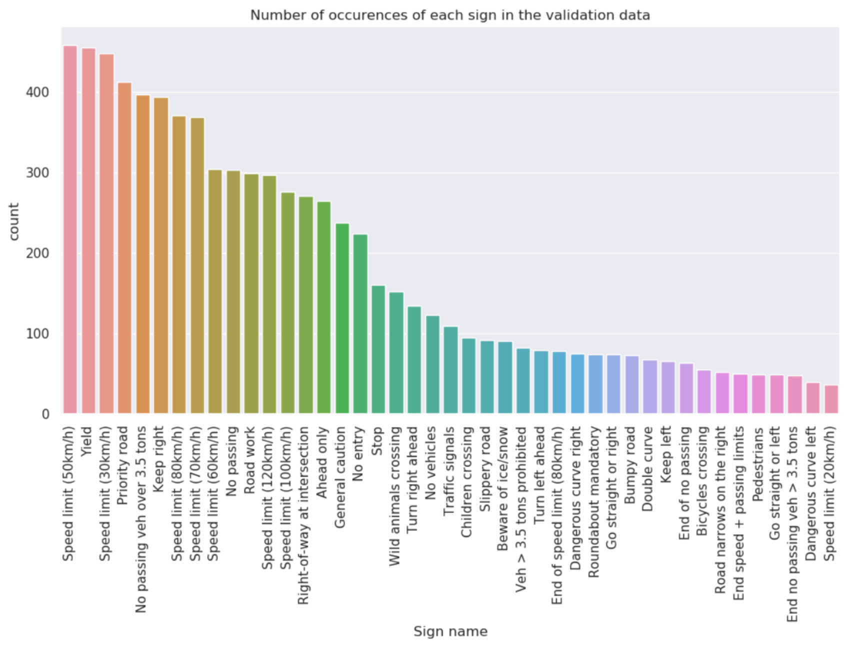

Figure 5: Number of occurrences of individual signs in validation data

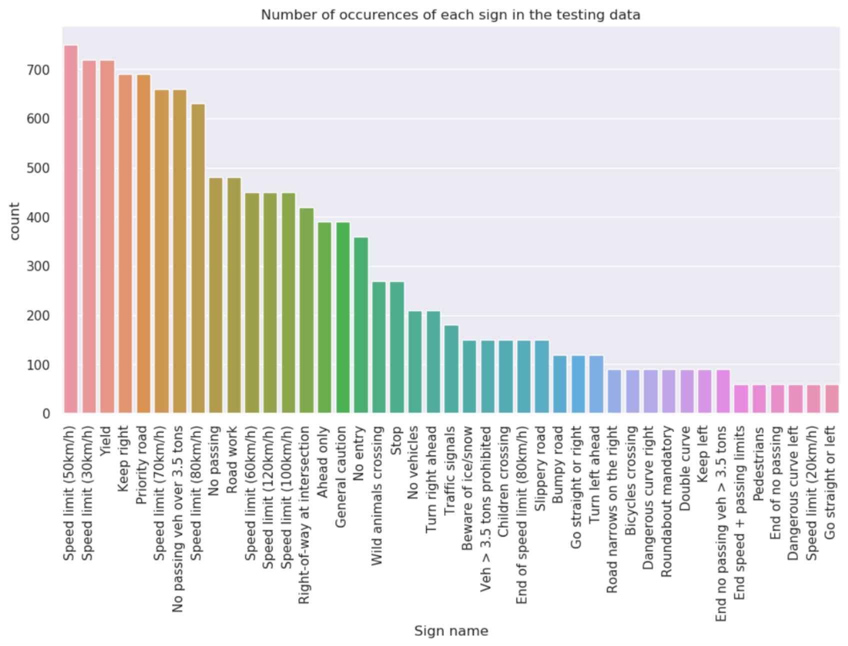

Figure 6: Number of occurrences of individual signs in testing data

Algorithms and Techniques

To solve this problem I used convolutional neural networks (CNNs). Compared to other neural networks which only contain fully connected layers, they are more suited to the processing of images. This is due to the fact that normal neural networks would become very huge with many parameters and that they are not as able to detect shapes and other attributes of images, as they have to flatten the images to a 1D vector.

CNNs accept a 2D matrix as an input and try to extract information from pixels that are near each other. This is achieved with different layers compared to a normal neural network. To actually use CNNs to solve a multi-class classification it contains fully connected layers for the output of the network.

One new layer is the convolutional layer. It works with a quadratic kernel which is basically a matrix of different weights. This kernel is moved over the input of the layer and affected pixels of the kernel are multiplied with the weight and combined into one output value. To reduce the dimensionality of the data there is also a pooling layer. It also works with a kernel, but the weights of this kernel are all 1. The afected inputs of the kernel are then combined using a predefined function. The most common one is max pooling, which has the maximum value of all inputs as the output.

A convolutional layer has the following hyperparameters: - Kernel size: How big is the quadratic convolutional windows. - Number of filters: The filters represent different patterns that get detected in a layer. More filters can detect more patterns. - Stride: The number of steps the convolutional window moves. - Padding: Can be used to make sure that the input of the layer has the same size as the output.

A CNN can be influenced by the number of layers and the type of these layers. We can also influence the way it is trained. We might change the number of epochs, which influences how often the dataset gets passed through the network in the training phase. Another value is the learning rate, which influences how fast the parameters in the layers are adjusted. A way to try different values for the number of epochs and learning rate is called hyperparameter tuning.

As the training of huge models is very time and cost intensive (resource cost) I made use of the transfer learning technique. There are common neural networks that were already trained on huge datasets by other people and companies. I can use these pretrained networks to solve my own problems by simply freezing the parameters of them and simply replacing the output layers for my use case. In the training step only the output layers will be trained which reduces the complexity immensely.

In my specific usecase I will use a ResNet50 CNN model. It was trained on the ImageNet dataset. They come without their output layers, so I will add a final fully connected layer as the output with either 43 or 6 output neurons, depending on the problem. These networks will then be trained with my given datset. This will represent a baseline for each of the solutions. Finally I will also use hyperparameter tuning to adjust the default values for training of my CNN to improve the results of my neural network.

Benchmark

The main benchmark that I used for my models is the best performing solution from the paper. It is the method cnn_hog3 from the team IDSIA. It achieved an Accuracy value of 98.98% for the combined classes approach. Its main problems lie in the recognition of mandatory and danger signs, where it has an accuracy of 97.89% and 98.83% respectively.

My main goal was not beating the performance of this model, but seeing how close a models can come today with minimal effort. This is especially important, as the model from the benchmark was heavily optimized by the team, hopefully showing that even non-experts can achieve scores close to them just 10 years later.

The approach for the recognition of all distinct signs does not have a comparable benchmark.

It is just to show how good the same approach is for a problem with way more classes.

III. Methodology

Data Preprocessing

The main data preprocessing lies in organizing the data and splitting the training data into training and validation datasets.

The structure of the data downloaded from Kaggle for the training data is, that each respective sign is in an folder corresponding to its class. To make the splitting easier, I started by moving all the files to the same folder and then randomly selecting the data which will be chosen for the validation and for the training. I chose to make a 80/20 split, where 80% of the data was used for training and 20% of the data was used for validation.

As shown in the plots in section , the distribution of classes between the different datasets is pretty similar, which means we don't have a huge skew of the data due to the splitting of the original training data.

In accordance with this splitting I also created different ground truth files, which only contained the information for the corresponding datasets.

Additionally the ground truth only included the classes for the distinct sign recognition approach. For the combined sign group approach I had to reconstruct the mapping from sign class to sign group according to the paper. This new ground truth was then also saved in corresponding files for training use.

Due to this new structure I needed to create a new dataset class for my dataloader. This is to make sure that the dataloader can easily access each image and get its corresponding label from the ground truth file, as it does not know the label from the folder it is located in anymore.

The only other preprocessing happens in the transformation function from the dataloader. Due to the fact that the size (width x length in pixels) varies for each image, we transform each image to have the same size of 224 x 224 pixels.

Implementation

The final neural networks are based on a transfer learning approach. The nets are made up of a ResNet50 backbone with frozen weights and the final classification happens in two fully connected layers: the first has the output of the backbone as the input and includes 128 neurons. This layer is connected with a ReLU activation function to the final layer which contains either 6 or 43 neurons, depending on the target task. The combined approach has 6 neurons while the distinct sign approach has 43.

I did use some specific hyperparameters for my base model. This corresponded to a learning rate of 0.001, a batch size of 16 and 1 epoch of training for each model.

Refinement

The main refinement happened with the hyperparameters. I used a hyperparameter tuning approach to finetune all three chosen parameters. In the following code snipped I have shown the used ranges for my tuning.

```
hyperparameter_ranges = {
    "lr": ContinuousParameter(0.001, 0.1),
    "batch-size": CategoricalParameter([16, 32, 64, 128]),
    "num-epochs": CategoricalParameter([1, 3, 6])
}
```

The final results for the combined approach are as follows:

- learning rate: 0.0553
- batch-size: 16
- number of epochs: 3

Accordingly for the distinct approach:

- learning rate: 0.0473
- batch-size: 128
- number of epochs: 6

With these settings the results improved by huge amounts, as shown in the following table.

Model approach	Accuracy
Base combined	59.5%
Base distinct	21.4%
Final combined	96.4%
Final distinct	62.6%

IV. Results

Model Evaluation and Validation

The main results of the models are already shown in section . Both final models used a higher number of epochs and a way higher learning rate compared to the baseline models. The learning rate has to be high so that the model can actually make the necessary changes in the low number of epochs it has available.

The results for each class are shown in the following figures which represent the confusion matrix for each model when applied on the test data. One can see that the base line models are still very faulty, while the final models have way better results.

The robustness of the models is guaranteed due to the usage of separate training, validation and testing datasets. Also as I used hyperparameter tuning I also made sure to use the best parameter for my models.

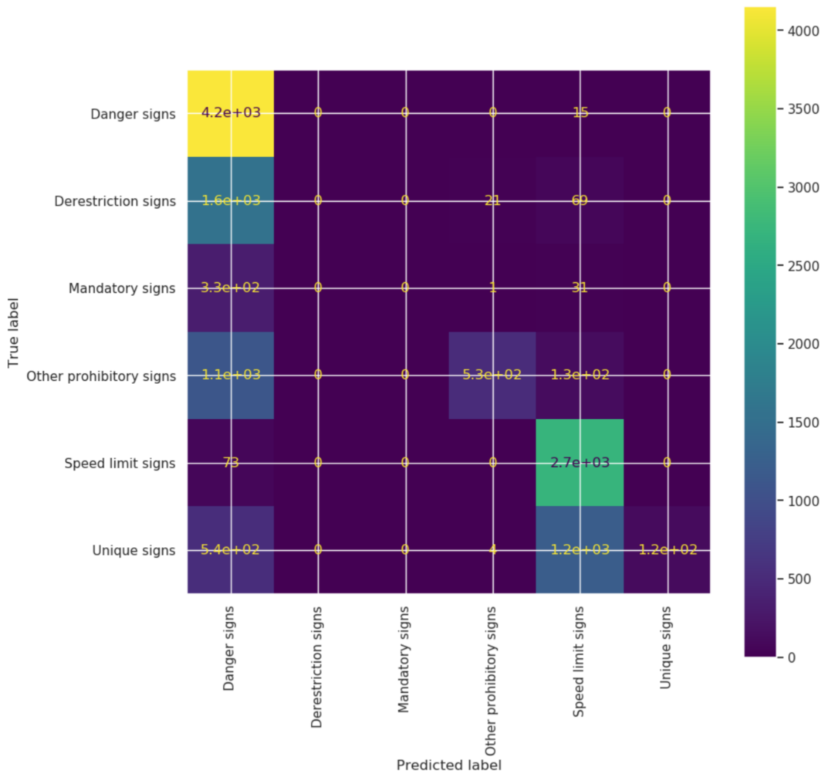

Figure 7: Confusion matrix of the base model with the combined sign group approach

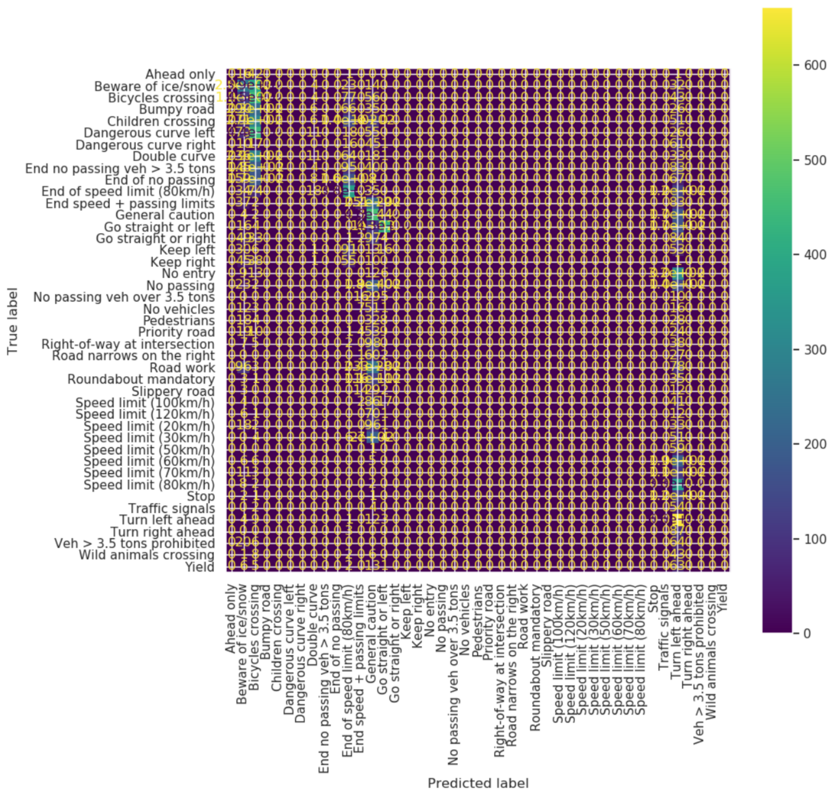

Figure 8: Confusion matrix of the base model with the distinct sign approach

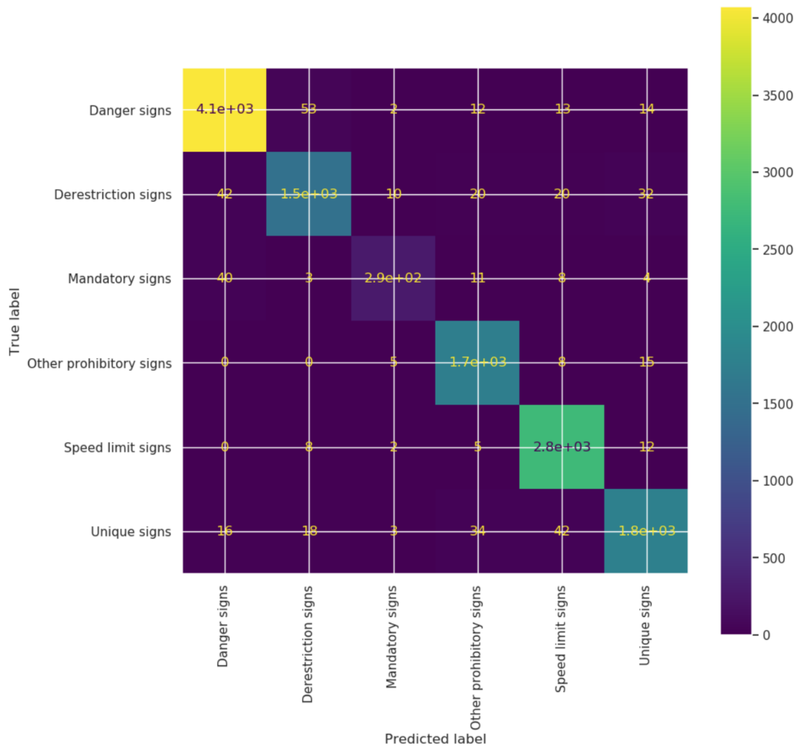

Figure 9: Confusion matrix of the final model with the combined sign group approach

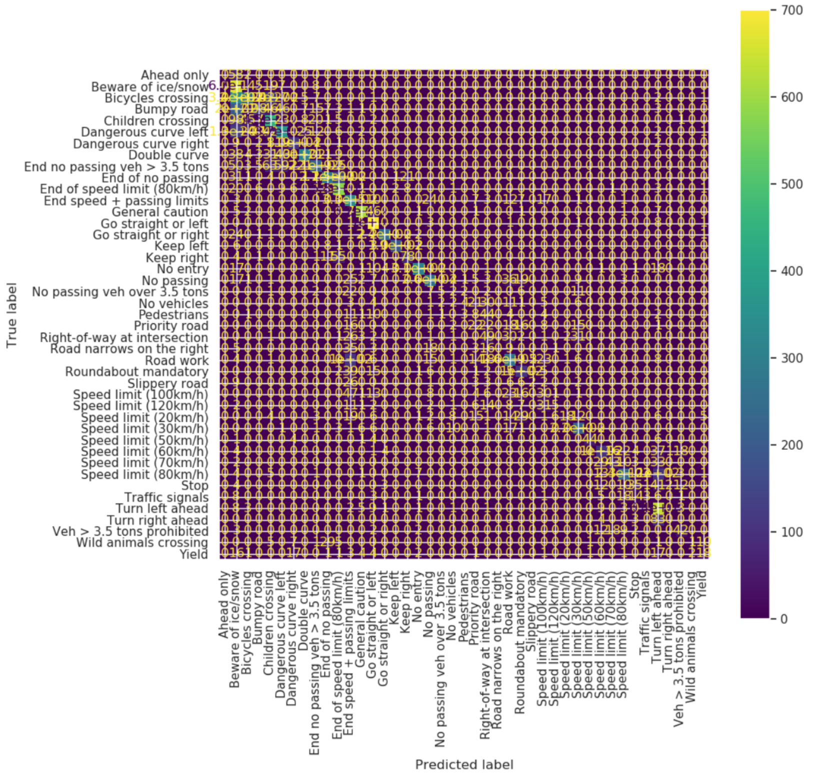

Figure 10: Confusion matrix of the final model with the distinct sign approach

Justification

The final solution model for the combined approach has an accuracy of 96.4%. While that is way higher than the baseline model with an accuracy of 59.5%, it falls a few percent points behind the benchmark with an accuracy of 98.98%.

The final solution model for the distinct approach has an accuracy of 62.6%. It is therefore lower due to the higher complexity but definitely beats my baseline with an accuracy of 21.4%.

While it does not beat the benchmark, it gets very close and shows just what is possible with a few optimizations.

V. Conclusion

Reflection

The whole end-to-end process to create a problem solution can be summarized as follows:

1. Define a problem for a given public dataset.
2. Find a suitable metric to determine the performance of the solution.
3. Download the data, format it to a certain structure and split the training data into training and validation sets.
4. Create the ground truth files for the given problem.
5. Create a custom dataset for the loading of the data.
6. Implement a simple CNN with a ResNet50 backbone with random hyperparameter values.
7. Tune the given hyperparameters and train the same CNN with the finetuned hyperparameters.

8. Calculate the chosen metric for the trained models to compare performance.

9. Deploy the models to an AWS endpoint and test it.

The most interesting aspect of the project was how easy it is to create models which perform similar to the best models from 10 years ago. Even though I had a limited amount of training time and computing resources, I still managed to create a model that was just a bit worse then the benchmark, even though the benchmark was highly optimized and created by a team of experts.

My approach to then create a model which is more complex due to having to detect each traffic sign class individually showed even more how impressive approaches using transfer learning can be. It enables everyone to create good performing models with a minimal amount of effort.

One challenging aspect of the process was to create the ground truth files and also splitting the datasets. Due to the fact that I had to move files it was often not easy to just rerun the same commands if something threw an error. I always had to make sure that the data is currently exactly in the expected format.

It was shown that my solution, using CNNs and transfer learning techniques, definitely fit my expectations for the problem. They are performing very well and nearly match the performance of the benchmark. While there are currently better performing solutions with transformers, they are generally way more computationally expensive and therefore most problems should most likely first try my approach for a given problem.

Improvement

There is definitely a lot of room for improvement, especially for the model for the distinct signs. Due to limited resources (limited performance due to cost) the hyperparameter ranges are chosen very small. Especially the number of epochs should be chosen way higher, more in the range of 50-200, to lead to better results. The hyperparameter tuner also only considered a few different combinations, so testing more might also be way more beneficial.

I also decided to freeze the parameters of the backbone to limit the training time. In a new approach I would not freeze them and allow the algorithm to also make fine adjustments to these layers and not only my new output layers. This would make the approach more suited to the given problem.

With just these few adjustments the results could probably improve a lot, but lead to more resources used when training the model.

References and Notes

SSSI11. STALLKAMP, Johannes ; SCHLIPSING, Marc ; SALMEN, Jan ; IGEL, Christian: The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In: *The 2011 International Joint Conference on Neural Networks*. San Jose, CA, USA : IEEE, Juli 2011. – ISBN 978–1–4244–9635–8, 1453–1460