Traffic sign categorizers: classifying traffic signs with simple CNN and pre-trained network

Izabell Anna Járó[1] and Emmer Marcell[2] and Kristóf Benedek [2]

[1] *Budapest University of Technology and Economics, TTK, Medical Physics MSc, 2nd year*
[2] *Budapest University of Technology and Economics, TTK, Research Physicist MSc, 2nd year*

January 14, 2024

In the last few years, research on artificial intelligence (AI) has intensified. In this work, we will create two different artificial neural networks (ANN), one is a more simple convolutional one (CNN), while the other is a large pre-trained one, the YOLOv8, where we adjust the last few layers to our needs. With them, we will categorize different prohibitory traffic signs into six categories and measure the ANN's accuracy. The simpler CNN will learn on a dataset with ~ 4190 sample per category. The image will contain a zoomed-in shot of the sign in question. We feed the YOLOv8 network images, on which the objects that have to be learned will be marked with boundary boxes. In the case of the YOLOv8 model, we used 3685 images per category for learning. We will monitor the learning process and evaluate the accuracy of the ANNs'. While in the case of the simpler CNN, conventional accuracy measure is a good indicator, the YOLOv8 using Mean Average Precision (mAP) or Intersection over Union (IoU) are better standards for object detection.

1 Introduction

In this era of information technology and computational revolution, developing and enhancing automatization methods is of key importance not only in industry, research, and everyday life. In the last two decades, a massive amount of digitalized data has accumulated, hence the work to create capable artificial intelligence (AI) has begun. Industry-wise, an awaited application of AI is to create self-driving cars. While there are numerous distinct tasks, that such an AI must be capable of doing, one of them is to recognize and distinguish traffic signs and then take appropriate action. We aimed to build an algorithm to recognize and then categorize into six classes of prohibitory traffic signs. The six categories are presented in Fig. 1.

We took two different approaches to tackle the problem. One is to implement a simple Convolutional Neural network (CNN), that learns the different categories. Here we used already cropped pictures[1,2] The other way is to teach the last few layers of a well-known, pre-trained model, which is one of the current paramount in object recognition, the YOLOv8 model. In this case, we use street-view-like pictures, where we know the labels and the bounding boxes of the objects of interest[3].

There was an attempt to create a toolchain that would detect objects that have red color and then select only those that can be fitted with a circular or elliptic shape. Still, due to a shortage of time and the nontrivial question of how to efficiently select only true areas of interest and then label them, we discontinued that path. Despite not finishing, our attempt to implement this detection method can be found in `scripts/edge_detection.ipynb`

Besides implementing these two methods, we have done some preprocessing of the data, such as image augmentation, image size reduction to save space, and filtering by labels and categories. For data augmentation, we implemented our pipeline as well and we used

Figure 1: Category A - no right, left, or U-turn

Figure 2: Category B - speed limit (regardless of the indicated value)

Figure 3: Category C - road closed

Figure 4: Category D - no entry

Figure 5: Category E - no stopping, no parking

Figure 6: Category F - other types of prohibitory traffic signs

Figure 1: *The six different prohibitory traffic sign classes that our algorithm has to recognize.*

the `imgaug` Python package for the YOLOv8, where the bounding boxes also had to transform, and while our pipeline can be extended as well, it was much more convenient to use an already existing toolchain. You can find these supporting toolchains along with all our other codes, links to the databases, and the saved models on our GitHub page[4].

2 Implementing the CNN

In the case of the CNN categorizer, we chose images that were already cropped (see Fig. 2).

As mentioned, we used $\sim$ 4190 images split into training and validation sets. The source of the images is the Chinese Traffic Sign Recognition Database (TRSD)[2] and the German Traffic Sign Recognition Benchmark Database (GTSRB)[1]. TRSD contains over 4200 images divided into 58 categories, while GTSRB has over 50 000 images divided into 43 categories. We selected the categories we needed and made a union of those. To avoid the bias coming from having an

Figure 2: *Example images from the CNN's training set. The labels correspond to the different categories in Fig. 1.*

unequal number of training images in the different classes, we implemented a data augmentation pipeline. This algorithm is relatively complex because it applies rotation, translation, affine transformation, and brightness change to the images, resulting in higher diversity. It also generates images in a n^3 fashion, where n is the number of the input samples. Hence, a high number of augmented images can be achieved from a relatively small number of inputs. This pipeline can be found in our GitHub repository[4] in `scripts/data_aug.ipynb`. We took then the category with the highest number of samples and generated augmented images for the rest of the categories such that their samples with the augmented images matched the category with the highest sample number.

The results of the CNN are presented in the `categorizer.ipynb` on our GitHub page[4]. We have also written a small GUI that can load the saved model and after feeding an appropriate image it will predict its category. The GUI is also written in Python and was made following Shikha Gupta's tutorial[5].

As for results, we trained the model for 15 epochs with a batch size of 32. Hence the learning rate was relatively low. Both the accuracy and the loss functions reached high and respectively low values, signs of successful learning. We detailed the results in the `categorizer.ipynb`, here we only show the time evolution of the loss and accuracy functions and the confusion matrix.

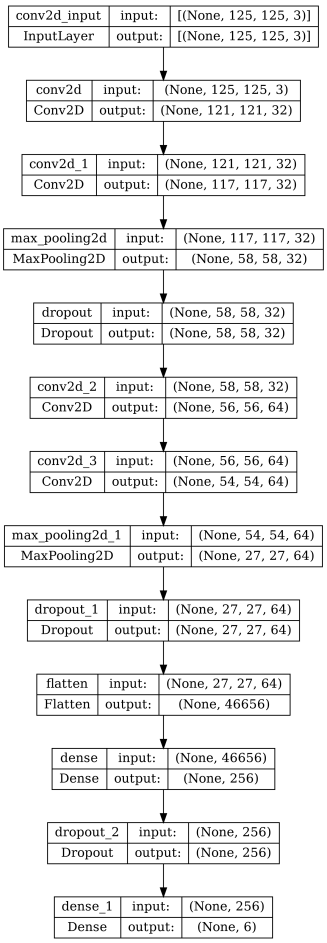

Figure 3: *The summary of the CNN. It has 6 layers, 4 convolutional and 2 dense, 2 dropouts, and 2 max pooling.*

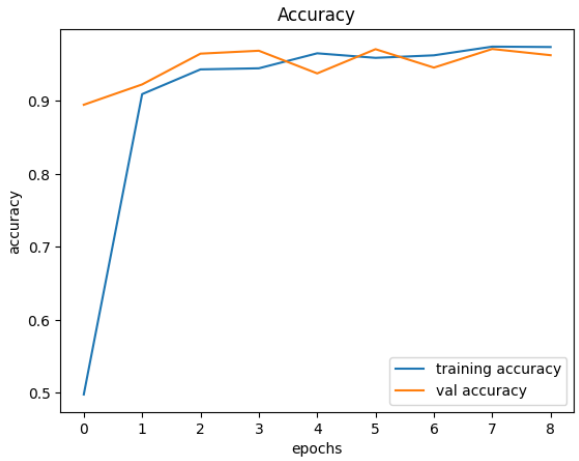

Figure 4: *Evolution of the accuracy function per epoch in the case of the simple categorizer CNN. For accuracy measure, we simply used if the predicted test label matches the actual test label. The accuracy reaches above 90% after a few epochs.*

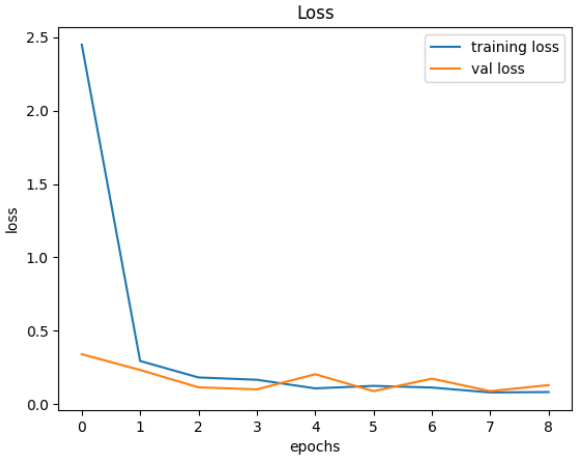

Figure 5: *Evolution of the loss function per epoch in the case of the simple categorizer CNN. For loss measure, we used the well-known categorical_crossentropy. The loss drops steeply, as expected, after a few epochs and the validation loss flattens around 5-6 epochs. One reason behind the low epoch number is the size of the dataset and the comparably low batch size.*

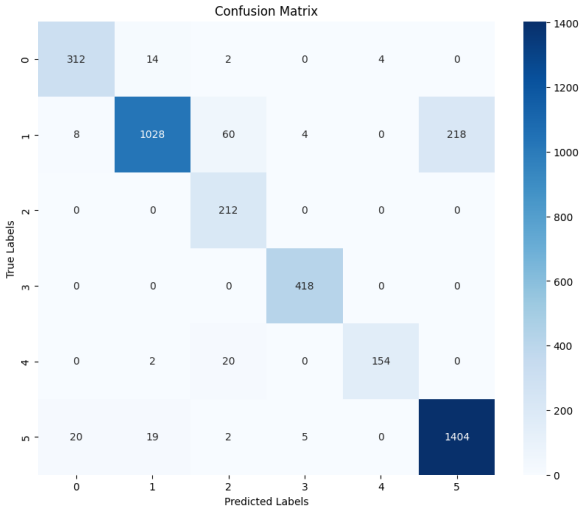

Figure 6: *Confusion matrix of the CNN for the test dataset. As expected in the case of successful learning, the diagonal elements of the confusion matrix are dominant. We can see that, label one, which corresponds to the speed limit signs, sometimes is predicted as the "other" category. Since this category also contains signs, that have writing and potentially numbers on them, can be understood. Another potential reason is, that we used both European and Chinese traffic signs and in some cases, these have minor differences.*

3 Implementing the YOLOv8

3.1 Data preprocessing

In the case of the YOLOv8, we used Google Streetview-like images. The images are of different resolutions

containing one or more traffic signs. The object's position is defined by its bounding box. While there are many possible sources to obtain such images, we have chosen Mapillary[3] as the source for our data, because this is one of the largest freely available datasets with fully annotated (labeled) high-resolution images.

Since the raw data is of 52 000 fully annotated images and these are of high resolution, the size of the source files is of 40 Gb. We wrote a script in Python with PIL and OpenCV to convert the images from their lossless format to lossy `.jpg` and reduced their size to $1/4$-th of the original. The trick in this code is that we left the resolution untouched, otherwise, we would've had to transform the bounding boxes as well, which is a more cumbersome task. After reducing the source files' sizes, we made a data frame from the `.json` annotations, and we selected the objects that fit into the six categories. After elimination, a sum of 13 550 objects remained on 9215 images.

The data package is already split into train, test, and validation sets, but the testing set has no labels. Since our code required labels for the testing set as well, because we wanted to check whether the answer of the code was correct or not and also wanted to select test images with objects that fit into our prohibitory sign categories, we used only the training and validation set, which we resplit into training, testing and validation part using the same ratio as before.

To ease the problem caused by the imbalanced data, we used data augmentation to have the same number of objects as in the largest class: 3685 objects. All of the other classes had more than 1000 elements except for class 'road closed' with its 260 objects, this way an excessive augmentation was needed for this class compared to the others. The data augmentation was performed using the `imgaug` Python package[6] with numerous transformations: random horizontal flip, affine transformations, gaussian blurring, sharpening, and changing the brightness and contrast. During augmentation, the transformation of the bounding boxes was also performed. The augmented images had a 4 times bigger size than the original size-reduced images, thus size-reduction was performed again on the augmented images with the same script mentioned above.

After this preprocessing, we ended up with 3685 images per category in the training set. Due to resource constraints, we have only used 1000 out of the 26 714 training images created by augmentation and 100 images for validation. The 20 images provided by the data challenge made up the test set.

3.2 Training the model

The model's training was done with the KerasCV library[7]. The already preprocessed data was converted to TensorFlow ragged tensors that are made for data with varying lengths, this is necessary since

Figure 7: *Example images from the YOLOv8's dataset. The red squares are the bounding boxes associated with different objects.*

the pictures can contain more than one traffic sign. As mentioned earlier, the yolo model was trained using transfer learning methods, starting from the `yolo_v8_s_backbone_coco` preset. This pre-trained model's weights were fine-tuned on the COCO dataset[8]. We froze the first 145 out of the 169 layers of the model, to speed up the learning process. This has left us with 14.028 trainable parameters. For the calculation of the mAP and IoU metrics, we have used the `PyCOCOCallback` from KeracCV. The tracking of our workflow was done with the Weights & Biases library[9]. The results of the runs can be found here. The results of the YOLO model we quite lackluster, since despite the over 7 hour training time it had only reached a mAP value of 0.000 74 on the test set. The model was trained for 100 epochs with an early stopping patience of 30. The training of the model is illustrated in figs. 8 to 9. Furthermore, Table 1 summarizes the resulting metrics on the valuation and test set.

4 Conclusions

Disappointingly, the promising YOLOv8 model, famous for its use in computer vision gave us lackluster results. This could be accounted for many reasons. Maybe the increase in training time on the images or the use of the full training set could have improved our results, but the time and resource constraints of Kaggle tied our hands. We are sure there must be a clear solution for training the model in a more rewarding manner and we plan to explore it in the future. A possible solution is to utilize the much better-documented `ultralytics` library that uses PyTorch for training. Although the preprocessing of our datasets is much more cumbersome

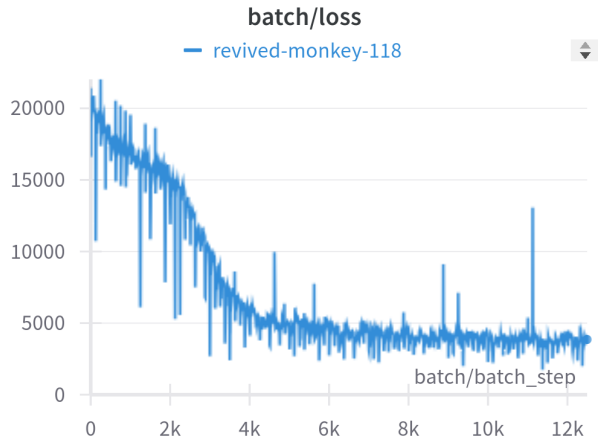

Figure 8: *Change of loss as a function of training steps.*

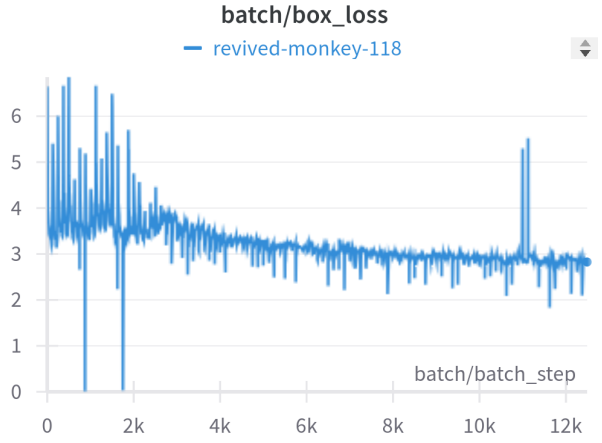

Figure 9: *Change of box loss as a function of training steps.*

in that case, that's why we chose `keras-cv` now.

On the other hand, we have shown that a simple CNN is capable of yielding reasonable results if the parameters and the structure are chosen well and the dataset is extensive enough. With relatively low computation cost, we achieved $\sim 92-95\%$ accuracy in the test dataset with our `categorizer.ipynb`. Of course, there is always the possibility to further enhance our model, i.e. to improve the accuracy in the case of label 1 (c.f. Fig 6.), but as a first go, it is a good result. As next ideas we could play around with the parameters more, changing the batch size and learning rate. Also in the case of the raw source files, there was a big inequality between the different categories, i.e., in some cases, we had to augment a given category much much more, than another.

epoch/val_AP	0.000025
epoch/val_AP50	0.000114
epoch/val_AP75	0
epoch/val_APs	0.000029
epoch/val_APm	0
epoch/val_APl	-1
epoch/val_ARmax1	0
epoch/val_ARmax10	0.00442
epoch/val_ARmax100	0.00481
epoch/val_ARs	0.00485
epoch/val_ARm	0
epoch/val_ARl	-1
test_MaP	0.000741
test_MaP@[IoU=50]	0.00495
test_MaP@[IoU=75]	0.000059
test_MaP@[area=small]	0.00125
test_MaP@[area=medium]	0.00266
test_MaP@[area=large]	0
test_Recall@[max_detections=1]	0.00318
test_Recall@[max_detections=10]	0.0343
test_Recall@[max_detections=100]	0.0478
test_Recall@[area=small]	0.0989
test_Recall@[area=medium]	0.03
test_Recall@[area=large]	0
_wandb/runtime	26982

Table 1: *Weights & Biases summary of the run revived-monkey-118*

Bibliography

[1] J. Stallkamp et al. "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition". In: *Neural Networks* 0 (2012), pp. –. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2012.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0893608012000457.

[2] *Chinese Traffic Sign Database*. URL: http://www.nlpr.ia.ac.cn/pal/trafficdata/recognition.html.

[3] Mapillary. *Mapillary Traffic Sign Dataset*. 2023. URL: https://www.mapillary.com/dataset/trafficsign/.

[4] Marcell Emmer, Izabell Járó, and Kristóf Benedek. *Traffic sign categorizers: classifying traffic signs with simple CNN and pre-trained network*. Version 2.0.4. Dec. 2023. URL: https://github.com/emmermarcell/AI-in-Data-Science-Data-Challenge.

[5] Shikha Gupta. *Traffic Signs Recognition using CNN and Keras in Python*. 2023. URL: https://www.analyticsvidhya.com/blog/2021/12/traffic-signs-recognition-using-cnn-and-keras-in-python/.

[6] *imgaug documentation*. 2020. URL: https://imgaug.readthedocs.io/en/latest/.

[7] Luke Wood et al. *KerasCV*. https://github.com/keras-team/keras-cv. 2022.

[8] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[9] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[10] Gitesh Chawda. *Efficient Object Detection with YOLOV8 and KerasCV*. 2023. URL: https://keras.io/examples/vision/yolov8/.