A House and a Car is All You Need

Aliaksandr Nekrashevich
Smith School of Business
aliaksandr.nekrashevich@queensu.ca

Abstract

This project is focused on the problem of traffic sign recognition, where it adapts and tunes the Capsule Network architecture for this particular classification task. The resulting architecture is evaluated on different benchmarks and explained with the LIME approach of Explainable AI. Moreover, this document has helpful tweaks and interesting findings.

1 Introduction

The automotive industry is essential for modern human society, improves personal mobility and simplifies access to jobs or services. With the development of autonomous driving, deep learning and vehicles are becoming connected closely. This naturally leads to traffic sign recognition and detection problems, which complement each other and can be considered as parts of the autonomous driving pipeline.

Moreover, car stories are very famous within the local Kingston community. In September 2021, for example, a Jeep Patriot SUV was driven into a house on Montreal Street in Kingston, ON. With enough force and momentum, the vehicle broke the walls, entered the residence and came to rest in the basement. Afterwards, the passengers said "We have to get out of here!", ran away and took off. In some sense, a house and a car are all they needed in this extraordinary and unexpected accident. To avoid this or similar situations, it is important to recognize car traffic signs effectively, and to carefully follow road motion guidelines. The story described has been mentioned in [Cro21], and together with the headline of the famous Transformer deep learning paper has motivated the title of this document.

2 Datasets

There are multiple publicly available benchmarks for car sign recognition on the Internet. The following databases were selected for evaluation and are described in detail below:

1. German Traffic Sign Recognition Benchmark (GTSRB). This dataset is probably the most popular benchmark in the area of traffic sign recognition and was introduced at the conference IJCNN 2011. The database consists of 50000 images, with 40000 dedicated for training and 10000 used for testing. GTSRB represents a large, lifelike dataset with 43 traffic sign classes, where the size of each image is fixed to 32x32. One can find and download the benchmark by the following link: `https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign`.
2. Chinese Traffic Sign Recognition Database (TSRD). This benchmark includes 6164 road sign images with 58 sign categories, where all photos were collected using BAIDU street-view cameras. The size and quality of the images vary, with 4170 images dedicated for training and 1994 images used at testing. One can find and download the dataset by the following link: `http://www.nlpr.ia.ac.cn/pal/trafficdata/recognition.html`.

Preprint. Under review.

3. Russian Road Sign DataBase (RSDB) is presented in two versions, where R1 contains 67 classes, and R3 has 106 categories. The size and quality of the images vary. The R1 version contains 25433 images for training and 7552 for testing. The R3 version is bigger, with 70688 images for training and 22968 images for testing. One can find and download the dataset by the following link: `https://yadi.sk/d/TX5k2hkEm9wqZ`.
4. Belgium Traffic Sign Dataset (BTSD) represents a multi-class classification problem into 62 different road sign classes. The benchmark is available by the following link: `https://btsd.ethz.ch/shareddata/`.

3 Brief Description

The idea of Capsule Networks was likely first proposed in work by [SFH17]. The point was to represent object types by capsules, assuming that capsule length represents beliefs of having a certain thing in a particular place. There are two types of capsules used: primary capsules capture crucial parts of the objects, and final recognition capsules capture beliefs of the specific capsule to be present in the picture. Final capsules typically correspond to the classes, and the longest capsule corresponds to the predicted class.

To turn on the idea of capsules, the aforementioned paper introduced the following squashing function: $v_j = \frac{||s_j||^2}{1+||s_j||^2} \frac{s_j}{||s_j||}$, where v_j is the vector output of capsule and s_j is pre-nonlinearity input. Essentially, this transformation is designed to shrink short vectors towards zero, and transform longer vectors into length slightly below 1.

Between capsule layers, the next capsule forms prediction vectors from the outputs of previous capsules by transforming them linearly. Afterwards, these prediction vectors are combined with coupling coefficients and squashed. To make combinations more efficient, [SFH17] introduces a routing algorithm that recomputes coupling coefficients.

Another peculiar tweak that made CapsNet possible is the usage of margin loss for class existence, where the class label is encoded with 0.9 and 0.1 instead of 1 and 0 correspondingly. Non-existence is blended with a 0.5 ratio, therefore penalizing more towards the correct class. As a regularization, the loss is blended with a reconstruction quality of the initial image from the capsule layers. The original scaling factor was selected as 0.0005.

4 Implementation Details

A natural first step with neural network models is to look for tutorials and existing implementations. This simplifies architecture design, but one needs to remember that many deep learning implementations available on the Internet have mistakes. Indeed, humans make errors, and GitHub repositories frequently lack the quality of funded and regularly updated libraries. There are multiple tutorials describing Capsule Networks on the Internet; here are the most relevant from those studied:

1. Understanding Hinton's Capsule Networks. `https://pechyonkin.me/capsules-1/`, `https://pechyonkin.me/capsules-2/`, `https://pechyonkin.me/capsules-3/`, `https://pechyonkin.me/capsules-4/`
2. Understand and apply CapsNet on Traffic sign classification. `https://becominghuman.ai/understand-and-apply-capsnet-on-traffic-sign-classification-a592e2d4a4ea`
3. Beginner's Guide to Capsule Networks. `https://www.kaggle.com/fizzbuzz/beginner-s-guide-to-capsule-networks`
4. CapsNet: Tensorflow implementation. `https://www.kaggle.com/giovanimachado/capsnet-tensorflow-implementation`

The following implementations were used as a starting point:

1. Pytorch Implementation from repository `https://github.com/higgsfield/Capsule-Network-Tutorial`
2. PyTorch Implementation from repository `https://github.com/cedrickchee/capsule-net-pytorch`

3. PyTorch Implementation from repository `https://github.com/adambielski/CapsNet-pytorch`
4. TensorFlow implementation from repository `https://github.com/XifengGuo/CapsNet-Keras`
5. TensorFlow implementation from repository `https://github.com/thibo73800/capsnet-traffic-sign-classifier`
6. TensorFlow implementation from repository `https://github.com/naturomics/CapsNet-Tensorflow`

After studying initial approaches available on the Internet, the PyTorch framework from `https://pytorch.org/` was selected for project implementation. It is typically easier to use PyTorch for drafts and prototyping, while Tensorflow `https://www.tensorflow.org/` is more stable and reliable in production and deployment.

5 Tweaks and Tricks

To make initial implementations work in the current setup, the following unexpected tweaks and fixes were applied:

1. It has turned out that original squash function is not ready to a big amount of classes, and therefore produce NaNs when the vector lengths come close to zero. The following modification was used instead: $v_j = \frac{(||s_j+\varepsilon_1||^2+\varepsilon_2)(s_j+\varepsilon_1)}{\varepsilon_3+(1+||s_j+\varepsilon_1||^2+\varepsilon_2)\cdot\sqrt{||s_j+\varepsilon_1||^2+\varepsilon_2+\varepsilon_4}}$. Although somewhat massive, numerical stability is better.
2. Agreement routing copied from some public repository was not batch-independent without explicit reasons (like training and testing mode in batch normalization). Therefore, it was moved to a particular torch.nn.Module and rewritten from scratch.
3. The Recognition capsule module copied from some public repository was rewritten to make it aligned with the agreement routing module.
4. Reconstruction Network copied from some public repository was taking softmax over the wrong dimension, which needed to be fixed. Interestingly, the prediction quality is still good enough but varies with the order when shuffling the test data loader.
5. Primary Capsules copied from some public repositories were slightly rewritten to introduce dimension alignment with new recognition capsules. The squash class method was moved to a separate module.
6. In the initial implementation copied from some public repository, the margin loss was not squared. However, it has turned out that other implementations on the Internet use it in a squared form. Therefore, this project also uses it in squared form by default. Optionally, there is an option to have it unsquared if needed.
7. On the Chinese dataset, it was clear that the initial model described in the paper overfits. Therefore, optional dropout was added to the Primary and Recognition capsule modules.
8. Optional batch normalization was added to original convolution for German, Chinese and Russian datasets. It makes training more stable on later epochs, allowing more reasonable constants in modified squashing.
9. When gradient behaves unstably, gradient clipping with norm 100 is implemented and applied.
10. If this gradient stabilization is not enough for effective training, NaN to zero gradient hooks are provided according to the manual available at `https://blog.paperspace.com/pytorch-hooks-gradient-clipping-debugging/`. Gradient hooks were added to RecognitonCaps and PrimaryCaps layers. These hooks help with training, but need to be removed when exporting graph to Tensorboard.
11. Data augmentation with randomly applied different affine transformations, hue-saturation modifications, and crops was implemented for all datasets. At the initial stage, the following GitHub repository provided the original idea: `https://github.com/joshwadd/Deep-traffic-sign-classification`.

12. The following tutorial was used to accelerate implementation and make the code work faster: `https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/c967c71b525d3cbe07b940373140aaef/tuning_guide.ipynb`. In particular, project implementation uses cuDNN benchmarking tweak, pinning memory in DataLoader, an additional worker for loading data with CPU while training on GPU, and torch.no_grad() on evaluation.

13. Tensorboard Support was added for visualizations. Checkpoints were added to track, visualize and explain the best model. Separate DataLoaders were implemented for each benchmark. LIME [RSG16] explanations were added to check if network output is reasonable.

14. An explicit call to garbage collection and cuda.clear_cache() were added after each epoch to make training possible on a batch with size 16. Problems occurred due to the 4GB video card memory limit on NVIDIA Quadro T2000 with Max-Q Design.

One can see the resulting network graph architecture in Figure 1. Typical configuration used images rescaled to 28x28, in grayscale, with 64 convolution channels from the Convolution Module. From the point of transformations, each image was augmented with a probability of 0.5. Each capsule at the PrimaryCaps module used 24 output convolution channels with output dimension $24 \cdot 6 \cdot 6$. The typical configuration contained 12 primary capsules; this choice is motivated by the limit of GPU-based memory. Recognition capsules were using 16-dimensional capsule outputs, and Reconstruction Net used two linear hidden layers of dimensions 512 and 1024.

Chinese benchmark was trained for 100 epochs, and other benchmarks were trained for 30 epochs. Batch normalization was used for Chinese, German and Russian databases. Dropout with a probability of 0.5 was used for the Chinese benchmark only. For the Chinese database, the augmentation probability was set to 0.6. Zero gradient hooks were implemented for the Russian dataset only. Typically, ε_i were of the order $1e-5$ and $1e-6$ for Belgium and Chinese benchmarks. German and Russian datasets used ε_i of the order $1e-4$.

Since deep learning represents black-box machine learning, a natural question is to try to explain what happens inside. This work explains the results through two major approaches. The first is model-specific and plots images reconstructed from the most relevant capsule. These images are naturally learned with reconstruction loss. Another approach towards explanations is LIME, which is described in the paper [RSG16]. This project applies the implementation example available at `https://github.com/marcotcr/lime/blob/master/doc/notebooks/Tutorial%20-%20MNIST%20and%20RF.ipynb`.

6 Numerical Results

Table 1 contains summary on the best launches. Experiments were performed on Dell Precision Workstation 5550 with 32 Gb RAM, Intel Core i7 CPU with six cores and 12 logical processors. Deep learning routines were executed on 4Gb NVIDIA Quadro T2000 with Max-Q Design.

Git repository with source code is available at: `https://github.com/smith-nekrald/car-sign-capsnet.git`. It contains README.md with launch instructions if the results want to be reproduced. The README.md also contains the link to download traindir with execution results and the link to download all benchmarks in one place.

	Benchmark	Test Accuracy	Train Accuracy	Test Loss	Train Loss	Epoch Id
0	germany	95.79	92.10	0.04	0.06	23
1	chinese	72.92	86.45	0.26	0.11	98
2	russian	93.83	93.39	0.05	0.06	20
3	belgium	99.85	90.71	0.00	0.08	22

Table 1: Experiment Summary.

Figure 1: Capsule Network Graph

6.1 Chinese Benchmark

6.1.1 Learning Curves on Chinese Dataset

Figure 2 demonstrates training accuracies and losses against training epochs. In Figure 3, one can find test accuracies and losses against training epochs. Running batch accuracies and losses are presented in Figure 4.

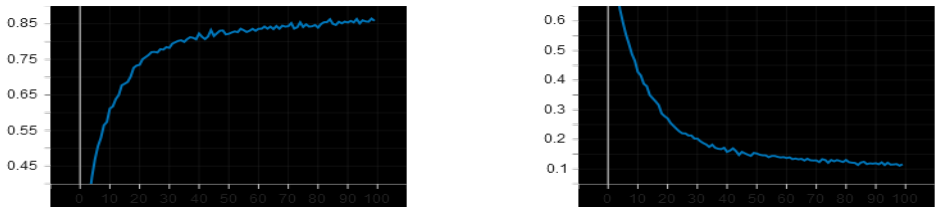

Figure 2: Chinese Dataset: Train Accuracies and Losses against training epochs.

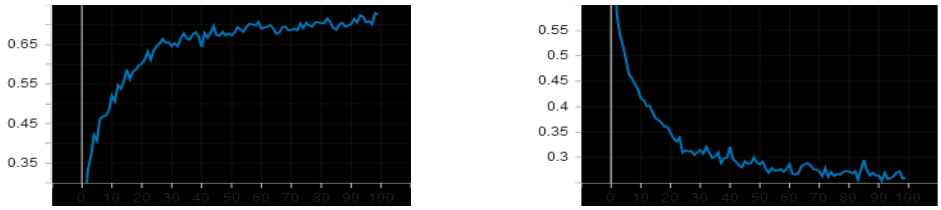

Figure 3: Chinese Dataset: Test Accuracies and Losses against training epochs.

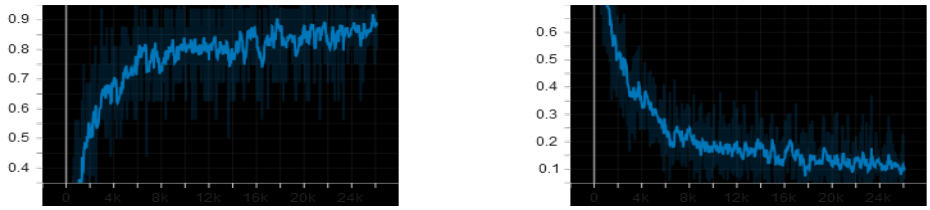

Figure 4: Chinese Dataset: Batch Running Train Accuracies and Losses against batch index.

6.1.2 Explaining Chinese Model

For the Chinese dataset, the reconstruction results are available in Figure 5.

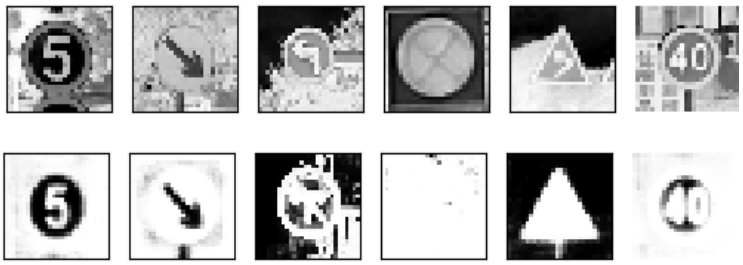

Figure 5: Chinese Dataset: Reconstructions from dominating capsules.

Explanations with LIME for the Chinese database are provided in Figure 6.

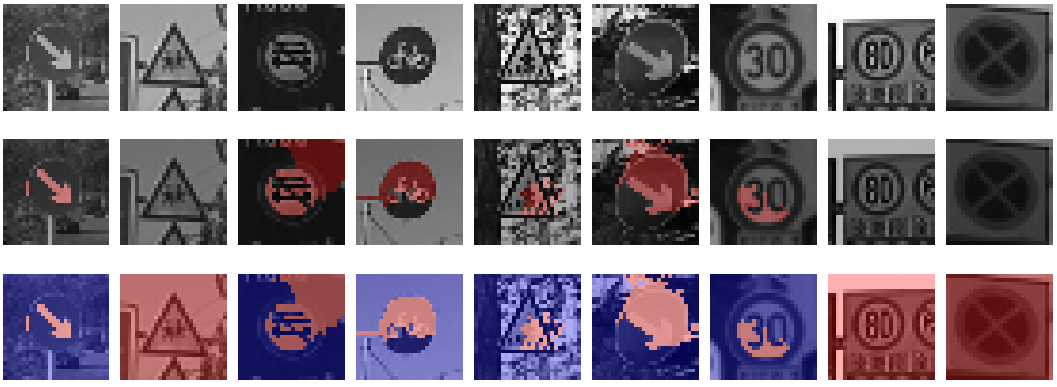

Figure 6: LIME Explanation for Chinese Dataset.

6.2 Russian Benchmark

6.2.1 Learning Curves on Russian Dataset

Figure 7 demonstrates training accuracies and losses against training epochs. In Figure 8, one can find test accuracies and losses against training epochs. Running batch accuracies and losses are presented in Figure 9.

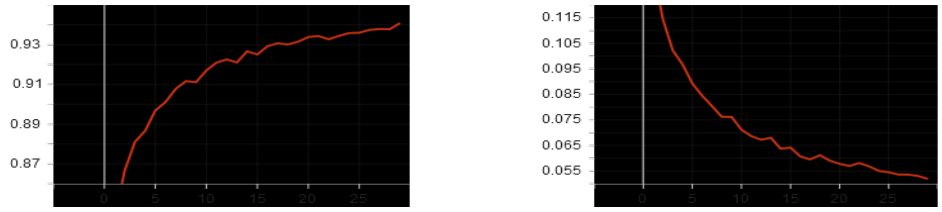

Figure 7: Russian Dataset: Train Accuracies and Losses against training epochs.

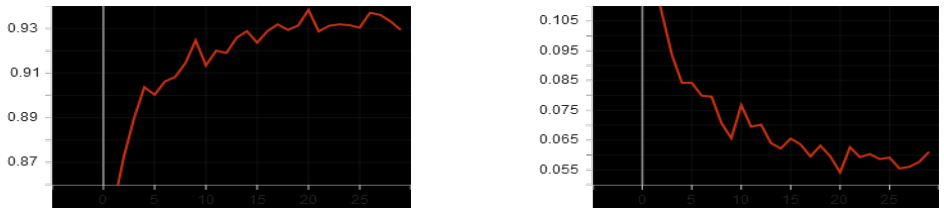

Figure 8: Russian Dataset: Test Accuracies and Losses against training epochs.

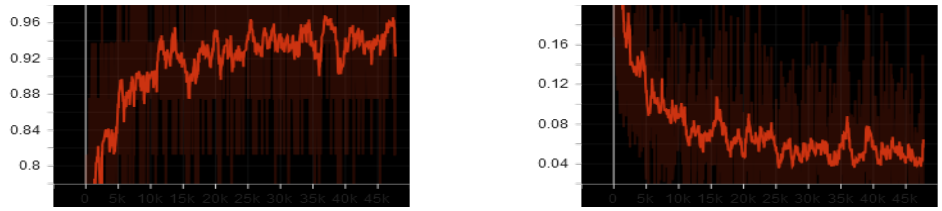

Figure 9: Russian Dataset: Batch Running Train Accuracies and Losses against batch index.

6.2.2 Explaining Russian Model

For the Russian dataset, the reconstruction results are available in Figure 10.

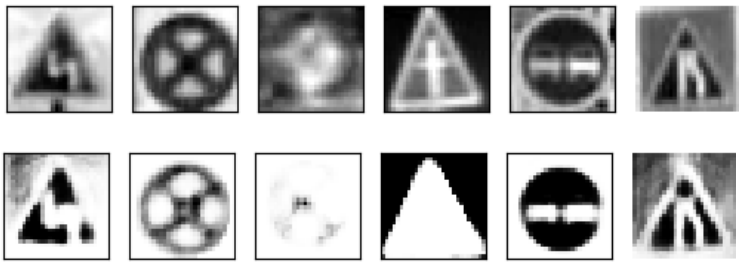

Figure 10: Russian Dataset: Reconstructions from dominating capsules.

Explanations with LIME for the Russian database are provided in Figure 11.

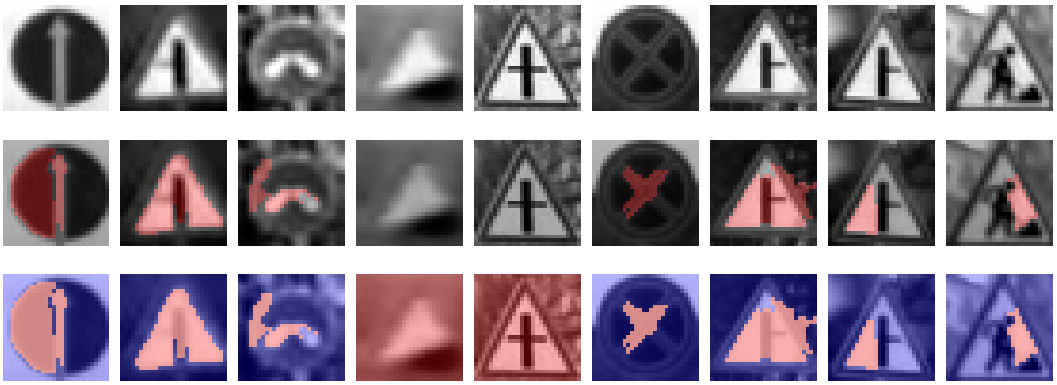

Figure 11: LIME Explanation for Russian Dataset.

6.3 German Benchmark

6.3.1 Learning Curves on German Dataset

Figure 12 demonstrates training accuracies and losses against training epochs. In Figure 13, one can find test accuracies and losses against training epochs. Running batch accuracies and losses are presented in Figure 14.

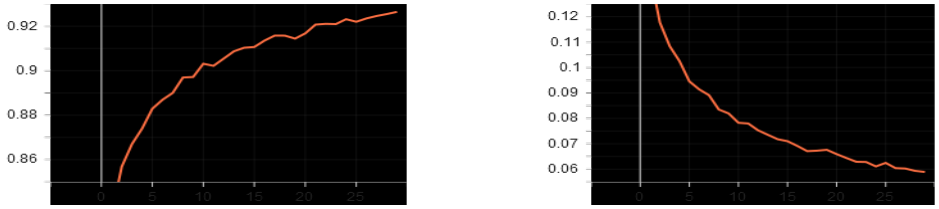

Figure 12: German Dataset: Train Accuracies and Losses against training epochs.

Figure 13: German Dataset: Test Accuracies and Losses against training epochs.

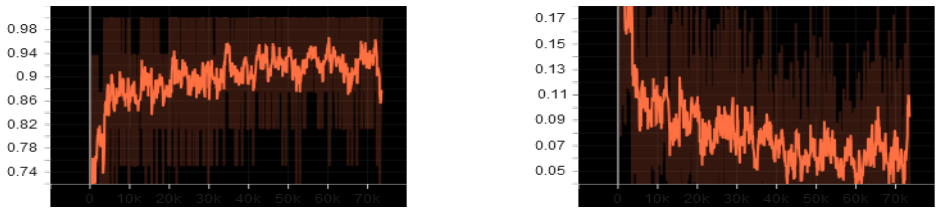

Figure 14: German Dataset: Batch Running Train Accuracies and Losses against batch index.

6.3.2 Explaining German Model

For the German dataset, the reconstruction results are available in Figure 15.

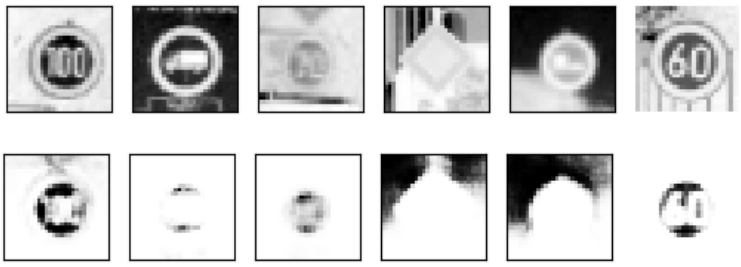

Figure 15: German Dataset: Reconstructions from dominating capsules.

Explanations with LIME for the German database are provided in Figure 16.

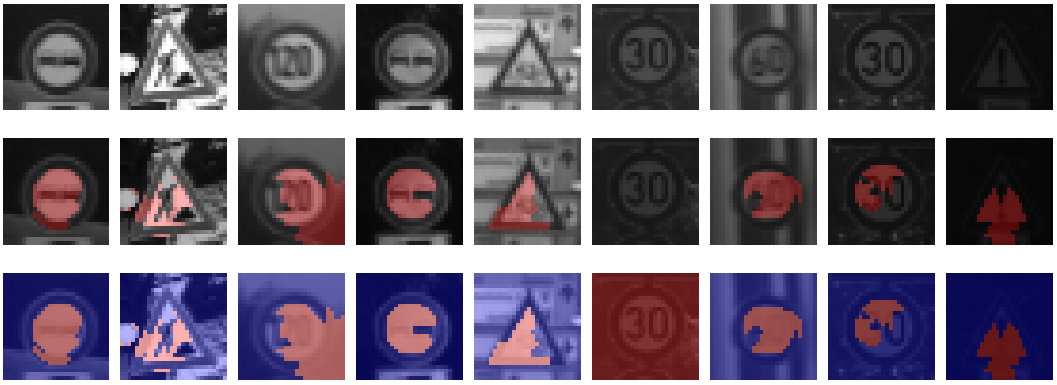

Figure 16: LIME Explanation for German Dataset.

6.4 Belgium Benchmark

6.4.1 Learning Curves on Belgium Dataset

Figure 17 demonstrates training accuracies and losses against training epochs. In Figure 18, one can find test accuracies and losses against training epochs. Running batch accuracies and losses are presented in Figure 19.

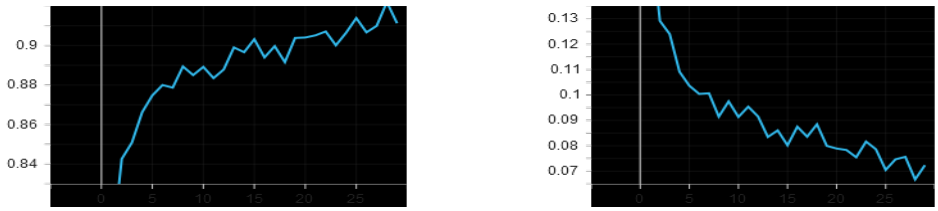

Figure 17: Belgium Dataset: Train Accuracies and Losses against training epochs.

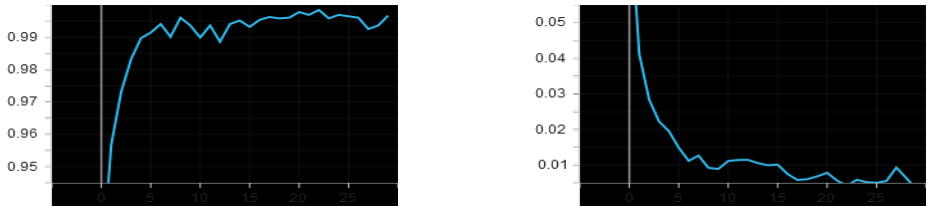

Figure 18: Belgium Dataset: Test Accuracies and Losses against training epochs.

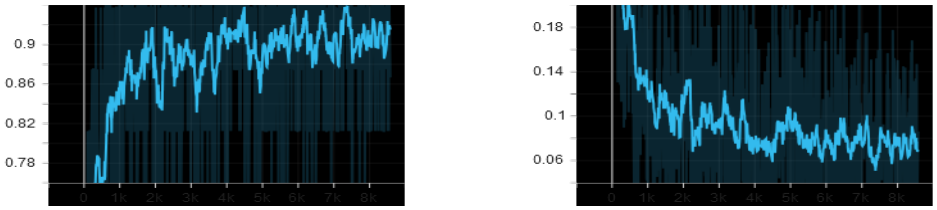

Figure 19: Belgium Dataset: Batch Running Train Accuracies and Losses against batch index.

6.4.2 Explaining Belgium Model

For the Belgium dataset, the reconstruction results are available in Figure 20.

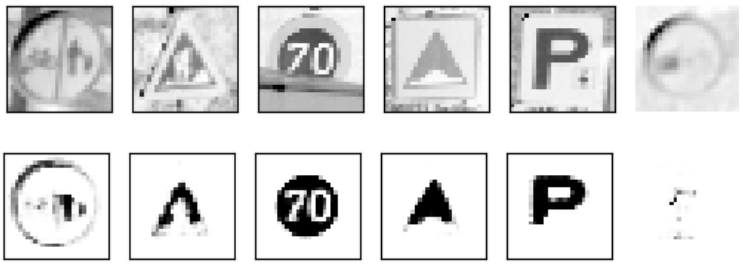

Figure 20: Belgium Dataset: Reconstructions from dominating capsules.

Explanations with LIME for the Belgium database are provided in Figure 21.

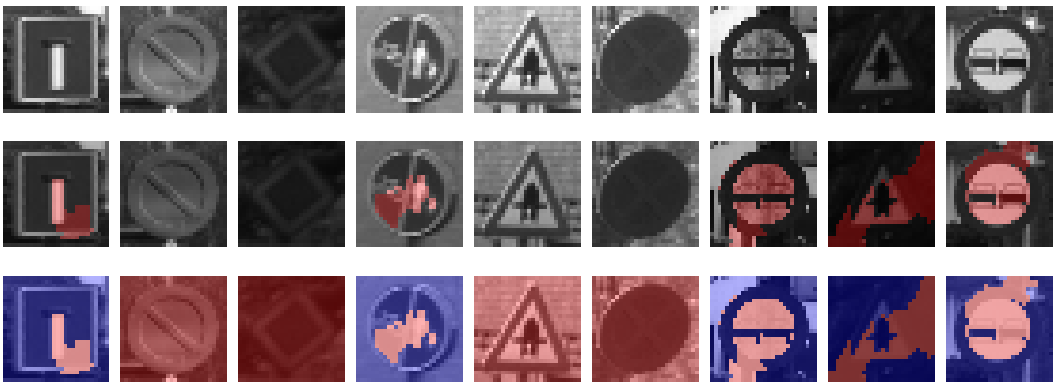

Figure 21: LIME Explanation for Belgium Dataset.

7 Conclusion

Traffic sign recognition is essentially a classification problem of categorizing car signs into corresponding classes. Understanding the signs can help to make better autonomous driving decisions, which can be done explicitly via traffic sign classification or implicitly with an end-to-end pipeline design. Despite the fact that the autonomous driving methodology has become really complex and manifold, the car sign recognition problem remains relevant and is often used for classification benchmarking.

References

[Cro21] Steph Crossier. Kingston police investigate after suv driven into house. *The Kingston Whig Standard, Online Edition, article issued on September 9*, 2021. Available at `https://www.thewhig.com/news/local-news/kingston-police-investigating-fail-to-remain-after-suv-driven-into-home`.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[SFH17] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. *arXiv preprint arXiv:1710.09829*, 2017.