Traffic sign detection and classification (with deep learning)

Eduardo Correia
up20186433
João Martins
up201806436
Ricardo Fontão
up201806317

FEUP
University of Porto
Porto, Portugal

Abstract

Recognizing traffic signs accurately is becoming more and more necessary, with the development of driver assistance systems and automated driving.

This process is usually composed of two main steps: classification of signs present in an image and detection of their spatial position.

The goal of this paper is to address a proposed solution, developed for the *Computer Vision* course, to the traffic sign detection and classification problem using deep learning computer vision methods.

1 Introduction

All models were trained and tested with the Python library PyTorch. Since the group developed two distinct solutions for the classification and object detection problems, their implementation was split into two different Jupyter Notebooks. This implementation can be found in Github. This also includes results from all trained models in a JSON format. All models can be found in this google drive.

2 Dataset

The provided dataset contains several photos containing one or more traffic signs, in varied situations. The available annotations only include traffic lights, stop signs, speed-limit signs and crosswalk signs.

This is a small dataset, containing only 877 images, which are then split into validation, train and test subsets.

Besides this, there are many images that contain signs that aren't represented in the annotations and there is a huge unbalance between class distribution (Figure 1). Speed limit signs alone represent about 70% of all the cases.

Given this, several techniques were employed to obtain a better performant model, such as data augmentation, optimizer tuning, learning rate scheduling, among others.

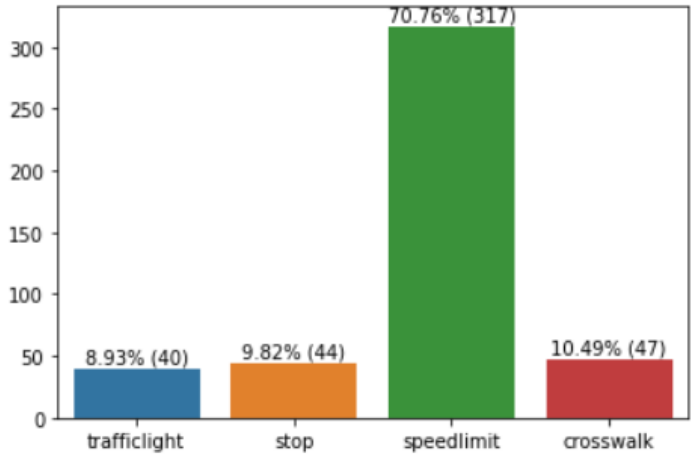

Figure 1: Class Distribution on Train Dataset

3 Traffic Sign Classification

This section will describe all models and architectures implemented for classifying traffic sign images, considering the problem as either multi-class or multi-label. For each model, all steps considered in order to find the best hyperparameters and results will be explained, with an evaluation of each model's performance.

Most of the models were trained during at least 10 epochs, with the train and validation subsets. To evaluate the results, all models were tested with the test subset, which contained all images specified for testing in the project's handout.

3.1 Multi-class Problem

In this problem, the model must identify the type of the biggest traffic sign of a given image. For this task, the dataset's annotations were used in order to assign each image with its respective biggest traffic sign type.

3.1.1 ResNet-50

The first trained model was a pre-trained ResNet-50 using fine tuning. All data was resized and normalized beforehand to the values that ResNet was trained to, i.e. size of 224 per 224, mean values of $[0.485, 0.456, 0.406]$, and std values of $[0.229, 0.224, 0.225]$. Aside from this, no other data augmentation techniques were employed. The following hyperparameters, resulting after some tweaking, were used:

- **Frozen Epochs** - 10
- **Fine Tuning Epochs** - 3
- **Optimizer** - SGD optimizer
- **Frozen Learning Rate** - 10^{-2}
- **Fine Tuning Learning Rate** - 10^{-6}
- **Loss** - Cross Entropy Loss

The used fine tuning process was simple: all layers except the last were frozen during the first 10 epochs, and afterwards all layers were unfrozen and trained for 3 epochs. The group found that performing a large number of epochs in this fine tuning phase would cause the model to overfit, scoring great results on the training data whilst scoring poorly on the test data.

This model was purposely made to be as simple as possible, as its results will serve as a baseline when discussing and comparing other models. The model results before fine tuning can be seen in table 1:

	🚦	🛑	🛑	🟦	Accuracy	Macro Avg.	Weighted Avg.
Precision	92%	100%	83%	90%	84%	91%	86%
Recall	50%	43%	100%	36%	84%	57%	84%
F1-Score	65%	61%	91%	51%	84%	67%	82%

Table 1: ResNet-50 - Pretrained Base Results

As can be seen, the model has 86% of weighted average. There is a clear bias on choosing speed-limits, and all other classes predictions have recall lower than 50%. This was expected, due to the unbalanced nature of the dataset.

Results after fine tuning can be found in table 2.

					Accuracy	Macro Avg.	Weighted Avg.
Precision	93%	100%	83%	90%	85%	92%	87%
Recall	58%	43%	100%	36%	85%	59%	85%
F1-Score	72%	61%	91%	51%	85%	69%	83%

Table 2: ResNet-50 - Pretrained Fine-Tuning Results

These metrics show that the model has learned how to better distinguish traffic lights, improving results from 9%. Precision values didn't seem to increase however, contrary to expectations.

The group also trained a non pre-trained ResNet-50 model from scratch. The used hyperparameters were the same as the ones used in the baseline model. This model performed according to table 3.

					Accuracy	Macro Avg.	Weighted Avg.
Precision	25%	29%	76%	0%	73%	32%	60%
Recall	8%	9%.	10%	0%	73%	29%	73%
F1-Score	12.5%	13%	86%	0%	73%	28%	65%

Table 3: ResNet-50 - Trained from Scratch Results

Training from scratch clearly shows worse results. The recall value of all speed-limits and crosswalks is almost negligible. This leads to a total accuracy of 32%. The group was expecting these values to be much higher. This might be due to the dataset not having sufficient varied examples to train the entirety of the network. The grouped tried to increase the number of training epochs and lowering the learning but the results remained roughly the same.

3.1.2 Optimizer Tuning

When defining the baseline model, the group tried several optimizers, with varying results for each one. In order to better compare and formalize these experiments, the previous model was trained with some of PyTorch's optimizers. These results can be found in 4.

	Weighted Avg
Adam	90%
Adamax	89%
RMS-Prop	88%

Table 4: Optimizer Results

All optimizers seem to produce better results, with the Adam optimizer improving results by almost 4%.

3.1.3 Learning Rate Scheduler

Similar to the last comparison, different learning rate schedulers were experimented using the same pretrained ResNet-50 model with transfer learning and with the SGD optimizer. The following table 5 showcases these results.

	Weighted Avg
Static LR	87%
Exp LR	89%
Step LR	90%

Table 5: Learning Rate Schedulers Results

These show that StepLR clearly produces better results. This is expected as manually tuning the learning rate in each epoch is infeasible and is prone to errors.

3.2 VGG-16

Similar to the Resnet-50 models, two VGG-16 networks were trained, one from scratch and the other pretrained with fine tuning. Since both the pretrained VGG and ResNet models were trained using images with the same dimensions and normalized transformations, the same dataloaders deployed for ResNet were used. The best schedulers and optimizers from the previous steps were used in this task. The following hyperparameters were used:

- **Frozen Epochs** - 10
- **Fine Tuning Epochs** - 3
- **Optimizer** - SGD optimizer
- **Frozen Learning Rate** - 10^{-2}
- **Fine Tuning Learning Rate** - 10^{-6}
- **Loss** - Cross Entropy Loss

These lead to the following results:

					Accuracy	Macro Avg.	Weighted Avg.
Precision	78%	94%	94%	50%	88%	79%	89%
Recall	88%	70%	94%	56%	88%	77%	88%
F1-Score	82%	80%	94%	53%	88%	77%	88%

Table 6: VGG-16 Results

As can be seen, VGG-16 has worse results than ResNet-50, with ResNet having 138 million parameters while VGG having 23 million. This is on pair with the Top-1 accuracy of each network, as described here [1]. There is a very small difference in performance between VGG trained from scratch and pretrained with fine tuning, which contrasts between the discrepancy found between the two ResNet networks.

3.3 Data Augmentation

As said previously, the used dataset is extremely small. For this reason, applying data augmentation to it will most likely improve results. Different combinations of data augmentation methods were tried, leading to the following transformations:

- Random scaling the image with a 20% probability
- Rotating the image a maximum of 40º with 20% probability
- Performing an elastic transform with 10% probability
- Flipping the image horizontally with 20% probability
- Downscaling the image with 10% probability
- Applying random brightness and contrast enhancement to the image with a 10% probability
- With a 20% probability, apply one of the following:
  - Applying Gaussian noise with 10% probability
  - Applying image compression with 20% probability
  - Sharpen the image with 30% probability
- With a probability of 20%, apply one of the following distortion filters:
  - Optical distortion with 30% probability
  - Grid distortion with 10% probability
- With a probability of 20%, apply one of the following blurring filters:
  - Motion blur with 30% probability
  - Median blur with 10% probability
  - Mean blur with 10% probability
- With a probability of 20%, apply one of the following color space filters:
  - Convert the image to grayscale with 20% probability
  - Convert the image to sepia with 20% probability
  - Shift the R, G and B channels between 40, 60 and 40 intervals respectively with 20% probability
  - Jitter the image's colors with 50% probability

A pretrained ResNet-50 with the Adam optimizer and ExpLR was used with the transformations above. This lead to much higher results , with an accuracy of 92% as was expected.

	🚦	🔴	🔴	🟦	Accuracy	Macro Avg.	Weighted Avg.
Precision	95%	100%	93%	63%	91%	88%	91%
Recall	83%	74%	96%	68%	91%	80%	91%
F1-Score	89%	85%	95%	65%	91%	83%	90%

Table 7: Data Augmentation Results

3.4 Custom Architecture

A custom architecture was developed initially to tackle the classification multi-class problem. Its diagram can be found in 2. The model was trained from scratch, using the same hyperparameters as the previous networks:

- **Epochs** - 10
- **Optimizer** - Adam optimizer
- **Frozen Learning Rate** - 10^{-2}
- **Loss** - Cross Entropy Loss
- **Learning Rate Scheduler** - ExponentialLR

The results for this architecture are as follows:

	🚦	🔴	🔴	🟦	Accuracy	Macro Avg.	Weighted Avg.
Precision	6.25%	14%	55%	9%	14%	21%	43%
Recall	46%	9%	11%	12%	14%	19%	14%
F1-Score	11%	11%	19%	10%	14%	13%	17%

Table 8: Custom Architecture Results

As can be seen, the custom architecture's performance was poor, with 42% accuracy. It can be concluded that this model struggles to identify almost all signs. The group was expecting this to be the case, as the designed architecture is very simple compared to the ResNet or VGG architectures.

3.5 Multi-label Classification

The multi-label problem consists in, for each traffic sign type, deciding whether the image contains a traffic sign of that type. For this purpose, the binary cross entropy function is used to take into consideration all classes when calculating the loss.

The three previous models (ResNet-50, VGG-16 and custom architecture) were adapted and trained to measure their performance in this problem. The best optimizers, schedulers were used alongside with the data augmentation transformations mentioned previously. This produced the following results 9.

	Macro Avg.	Weighted Avg.
ResNet-50	90%	89%
VGG-16	87%	89%.
Custom	18%	52%

Table 9: Multilabeling for all architectures

Surprisingly, VGG-16 and ResNet-50 performed equally, different from the multi class results. The group believes that this is due to the fact that VGG-16 is struggling to find the biggest traffic sign. In fact, almost all images in which VGG-16 struggled to identify the correct sign were images where the two biggest signs were reasonably the same size.

4 Traffic Sign Detection

The objective of an object detection task is to identify the position of any objects in an image and classify them. In our case, this implies finding all traffic signs and determining the minimum bounding box around its shape and classifying them according to the labels available in the dataset. Two approaches will be presented, a two-stage architecture using the Faster R-CNN architecture and a single-stage architecture using the YOLOv5 architecture.

4.1 Metrics

On top of predicting classes, object detection also outputs the bounding box in which it detects an object. To measure how well the model predicts the bounding box of a detected object, we will be using the Intersection over Union(IoU) metric also known as Jaccard Index. It is defined by the quotient between the intersection of the predicted bounding box and the ground truth and their union. The more the bounding boxes overlap, the higher the score will be.

Unlike previous sections, the metric to improve is not the accuracy but the mAP (Mean Average Precision) as defined by the COCO dataset [4]. This is calculated by averaging the Average Precision at different thresholds of IoU. The standard thresholds are 0.5 up to 0.95 with a step of 0.05, which totals 10 thresholds. The metric is then averaged across all 4 categories. Some other important values are the AP@IoU=.50 and AP@IoU=.75.

Due to resource constraints, unless otherwise specified, all models were trained across 10 epochs.

4.2 Two-stage Object Detection Architecture

On a first approach, a two-stage architecture was used for the object detection. On a general sense, the first stage consists of region proposal, in which the model selects which regions of the image contain objects. Those regions are then fed to the second stage, in which those respective objects are classified.

4.2.1 Architecture details

There were several architectures available, such as R-CNN, Fast R-CNN and Faster R-CNN [5]. We decided to choose the Faster R-CNN because of the performance improvements over its predecessors. Several backbones were available with pretrained weights, but the most interesting were the ResNet50 network and the MobileNet3 Large network. After training some models with the ResNet50 backbone, we deemed it too slow, so we decided to use the MobileNet3 backbone at the cost of some performance but faster training times.

4.2.2 Performance evaluation

The first step is to establish a baseline to which we can compare any changes we experiment. The baseline defined by us is the network trained across 10 epochs with its backbone frozen and then the entire network unfrozen for an additional 3 epochs.

Much like the previous sections, we mainly focused our experimentation in trying different optimizers and learning rate schedulers. On a first instance, we experimented with the SGD, Adam and Adamax optimizers. Since SGD was the best performing optimizer, we used it to test different learning rate schedulers. The schedulers tested were the ExponentialLR and StepLR. On this department, in conjunction with the Adam optimizer, the StepLR scheduler performed much better than the ExponentialLR seeing as this one lowered the score drastically. This is interesting when compared with the classification results mentioned previously, where the best optimzer was Adam and the best scheduler was StepLR. In all the runs, the learning rate started at 1e-2 and was reduced to 1e-6. When schedulers were used, the gamma was set at 0.9.

Similar to before, fine tuning with 3 extra epochs were performed. In this fine-tuning the backbone is frozen. However, the

results were very unsatisfying, only increasing the mAP metric in about 1% maximum. For this reason, it was decided not to distinguish the metrics before and after fine-tuning.

All results can be found in table 10. In the annex, some predictions made by the models can be found.

	Validation (Best)		Test	
	mAP	mAP@0.5	mAP	mAP@0.5
Baseline (SGD)	0.46	0.70	0.56	0.78
Adam optimizer	0.24	0.54	0.26	0.60
Adamax optimizer	0.37	0.63	0.46	0.78
StepLR (SGD)	0.41	0.66	0.55	0.80
ExponentialLR (SGD)	0.11	0.21	0.11	0.23

Table 10: Faster R-CNN Results

4.3 Single-stage Object Detection Architecture

For the single-stage architecture, the YOLOv5 network[3] was chosen. This choice came down to YOLOv5 being widely used and being easy to use. Unlike the two-stage architecture, YOLO only needs one pass on the image to produce the result. This is achieved by splitting the image in cells, which then each decide if an object is present or not, as well as what object it might be. Finally, all these results are joined and Non-Maximum Suppression is applied to leave only the interesting bounding boxes. This results in a significant speed improvement over the Faster R-CNN network.

Due to resource constraints, the chosen model was the YOLOv5s, the second-smallest available variant. This allowed us to train the models faster, but at the cost of some performance.

4.3.1 Dataset annotations

The dataset provided contained annotations in the PASCALVOC format, however the YOLOv5 model has its own type of annotations. The PASCALVOC format is defined in an XML format with the following format for each bounding box: (xmin, ymin, xmax, ymax), with all values in absolute coordinates. The YOLO annotation format is defined in a TXT file and has the following notation: (xmid, ymid, height, width) with all values as a percentage of the total height or width. A simple Python script was used to convert between both formats.

4.3.2 Performance evaluation

The first step was to train a baseline model to then make comparisons with. The baseline chosen is the pretrained model (COCO dataset) trained for 10 epochs with YOLO's default parameters. Once the baseline is established, we can play with the parameters, trying to improve the model's performance.

The first change done was switching the optimizer from SGD to Adam. This yielded poorer results, but the model maintained the climb of the mAP, so this was very likely limited by the small number of training epochs.

Following this change, we experimented with Test-Time Augmentation(TTA). Instead of showing a single image to the trained model, Test-Time Augmentation involves feeding several transformed images and then averaging the model's predictions. YOLOv5 provides this feature simply by using the *–augment* flag in the testing script.

All results can be found in table 11. In the annex, some predictions made by the models can be found.

	Validation (Best)		Test	
	mAP	mAP@0.5	mAP	mAP@0.5
Baseline	0.52	0.74	0.65	0.91
Test-Time Augmentation	-	-	0.64	0.91
Adam optimizer	0.37	0.51	0.44	0.64

Table 11: YOLOv5 Results

5 Conclusion

Future work might entail pretraining the custom architecture with a larger dataset, such as ImageNet (or a subset of it [2]), to train the first layers of the model and perform transfer learn with the problem's dataset. In addition, expanding the existing dataset in order to compensate for speed-limit imbalance may also improve results. The custom architecture may also benefit from shortcut connections in order to reduce the number of parameters of the network. This would allow for a higher number of layers that could, in turn, enhance the capability of the network to learn features of the dataset. The group also believes that with more powerful hardware available, the models could be trained faster and present much better results.

In conclusion, the group believes to have built an ample set of models with satisfactory results. The main difficulty encountered by the models in the multi-class problem was to find the sign with the biggest area. This might be mitigated with a larger dataset, that would provide the models with more opportunities to learn about these cases.

References

[1] Eugenio Culurciello. Neural network architectures, 2017. URL `https://towardsdatascience.com/neural-network-architectures-156e5bad51ba`.

[2] Ilya Figotin. Imagenet 1000 (mini, 2020. URL `https://www.kaggle.com/datasets/ifigotin/imagenetmini-1000`.

[3] Glenn Jocher. ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference, February 2022. URL `https://doi.org/10.5281/zenodo.6222936`.

[4] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014. URL `https://arxiv.org/abs/1405.0312`.

[5] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015. URL `https://arxiv.org/abs/1506.01497`.

Annex

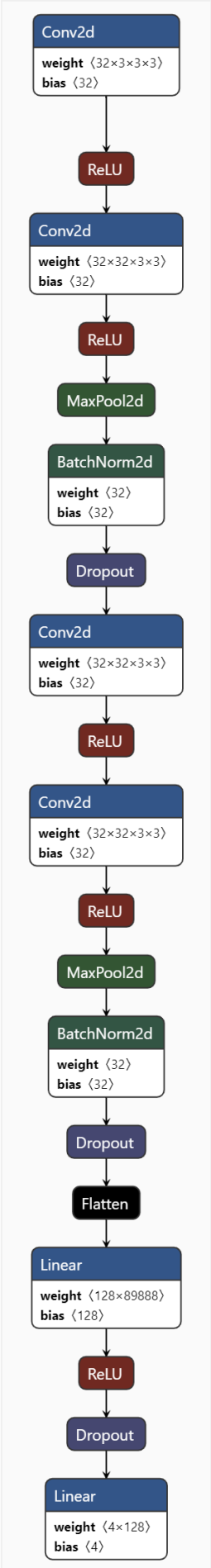

Figure 2: Custom Architecture Diagram

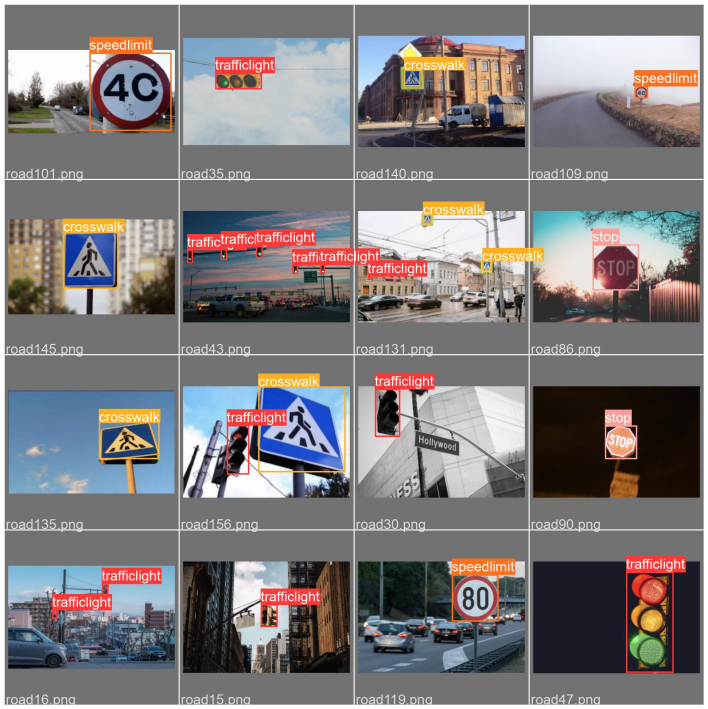

Figure 3: YOLOv5 ground truth

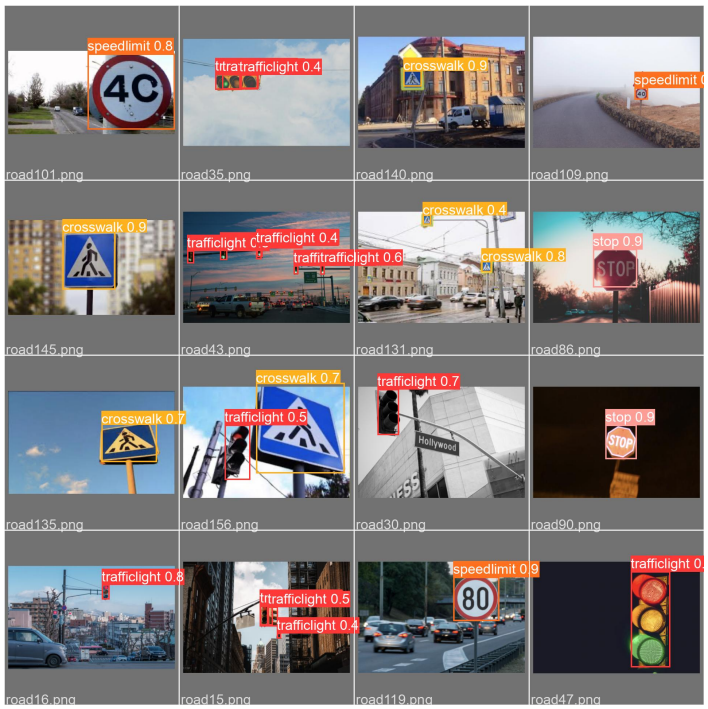

Figure 4: YOLOv5 baseline predictions

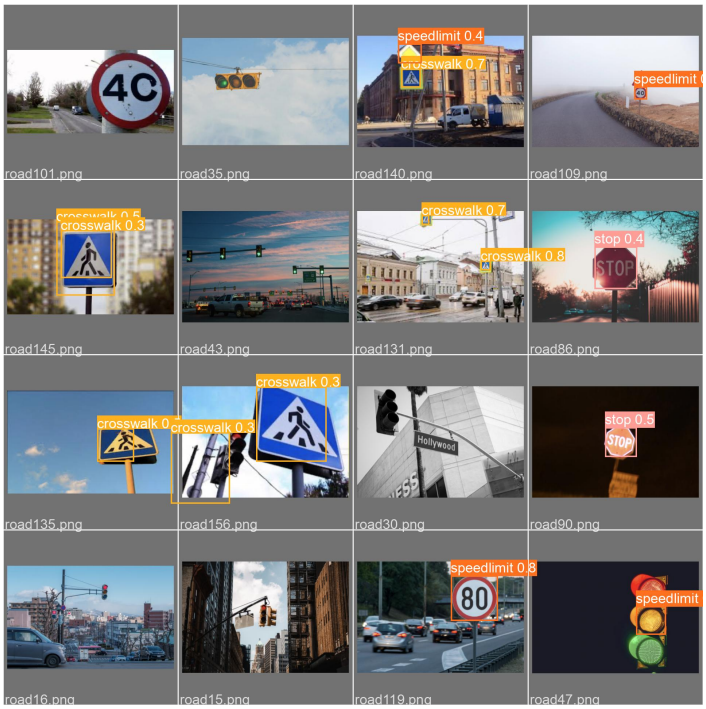

Figure 5: YOLOv5 Adam optimizer predictions

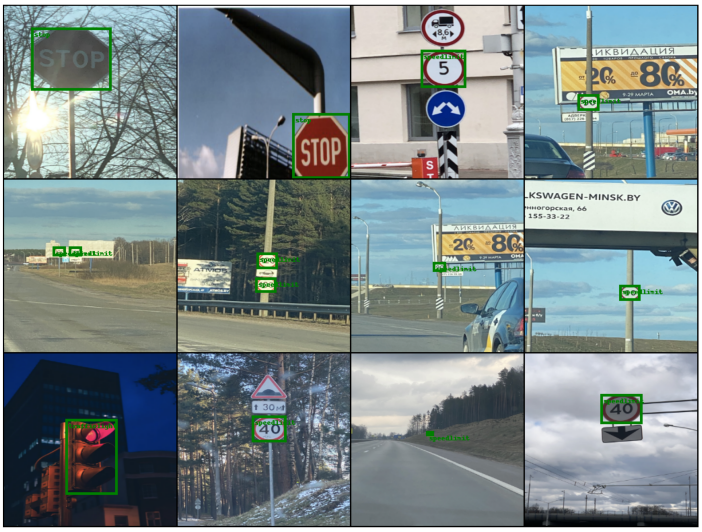

Figure 7: Faster R-CNN ground truth

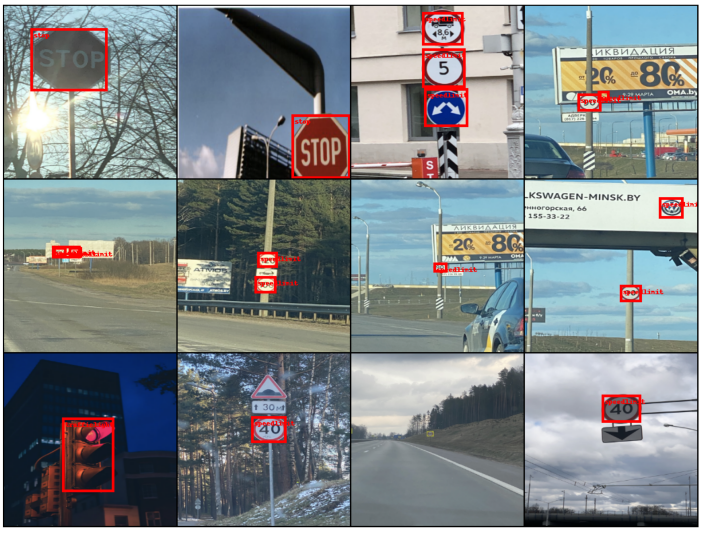

Figure 8: Faster R-CNN SGD preditions

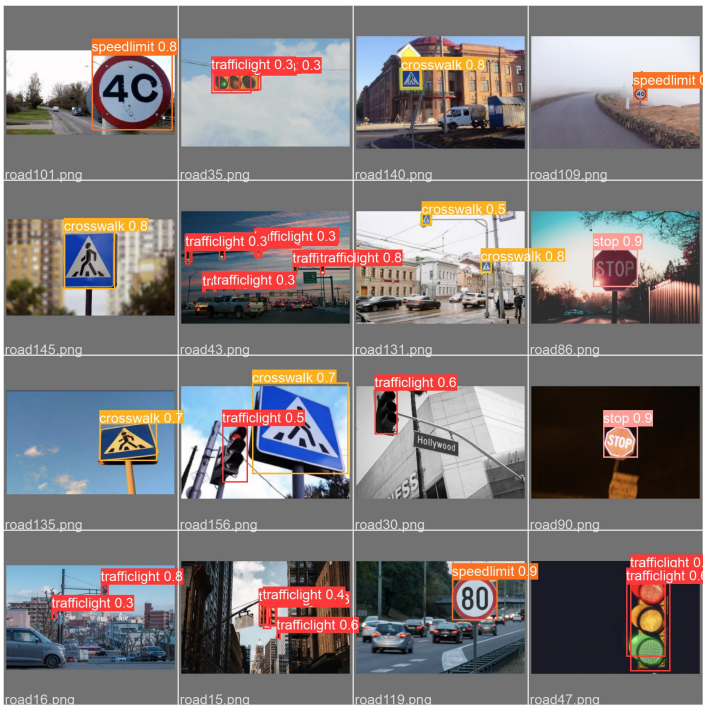

Figure 6: YOLOv5 Test-time augmentation predictions

Figure 9: Faster R-CNN Adam predictions

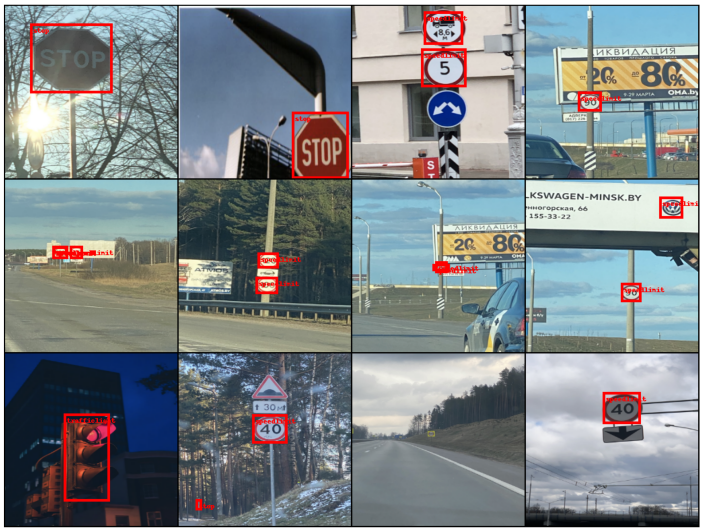

Figure 10: Faster R-CNN Adamax predictions

Figure 11: Faster R-CNN ExponentialLR predictions

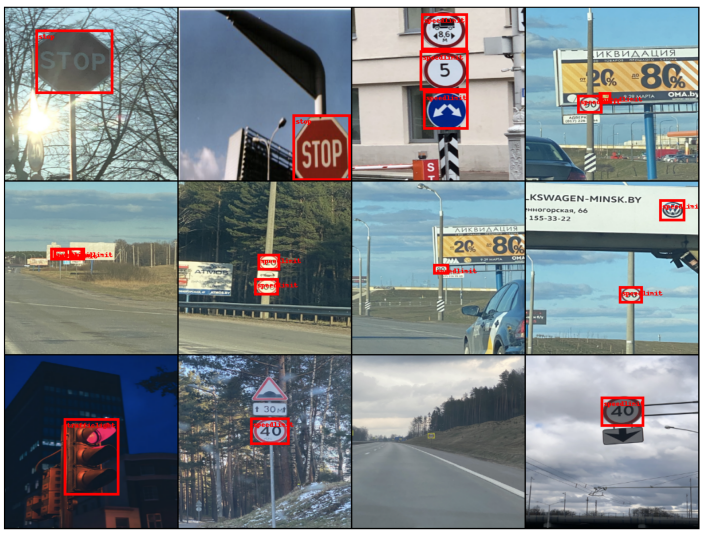

Figure 12: Faster R-CNN StepLR predictions