Traffic Sign Detection and Classification with Deep Learning

Diogo Samuel Fernandes
up201806250@up.pt
Paulo Ribeiro
up201806505@up.pt
Pedro Ferreira
up201806506@up.pt

Faculdade de Engenharia da
Universidade do Porto,
Porto, Portugal

Abstract

This second project focuses on the development of a program for the automatic detection and classification of a subset of traffic signs, namely traffic lights, stop signs, speed limit signs and crosswalk signs, using a Deep Learning approach. After the foundation of the implemented neural networks was laid, we began to adapt them to our problem, coming across numerous obstacles.

1 Data Processing

The dataset used for the traffic sign detection in this project was taken from the Kaggle dataset provided. From this set of images, a division in test and train subsets was gathered respecting the division indicated in the paper. Furthermore, we also opted to create a subset of the train images for validation purposes.

1.1 Train and Validation datasets

A class labelled *ImageClassificationDataset* was built to help with the various versions of image classification. This class enables us to construct our dataset based on the version to be utilised. This means that if the option for single-label classification is selected, as, in the basic and intermediate versions, only one label is returned. The chosen label is the one whose corresponding traffic sign occupies a larger area on the image. For this, we compare the sizes of each bounding box in the image (gathered from the annotations files) and retrieve the larger one. However, if the advanced version is selected, all of the image's labels are returned, not only the larger one.

The training dataset was divided into training (80% of the images) and validation (the remaining 20% of the images) to achieve better results. The training Dataset is a sample of data used to fit the model. Meanwhile, the validation dataset is the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. We can avoid overfitting issues by using the validation dataset.

1.2 Data augmentation

When we obtained the results of our first implementations of neural networks, we noticed that, despite the accuracy being reasonable, our model almost always predicted the label 2, which corresponds to the speed limit signs. We also verified that most of the images actually had this label as correct, which justifies the reasonable accuracy of the model. However, the model missed almost all the images in which the larger area signs were not speed limit signs. Therefore, we concluded that we were facing an overfitting problem, in which the model adapted too much to the training data, which could lead the model to fail future predictions. We also concluded that a possible cause of this problem would be the possibility of the training data being imbalanced, with large differences between the amount of samples for each label [1]. After verifying the distribution, our theory proved to be correct, having verified a great supremacy of images whose label was the speed limit signs in relation to the other existing signs: see figures 1 and 2. This was a serious problem, which limited the predictive capacity of our model, so we applied two different techniques to solve it, namely:

- Data Augmentation: We applied some techniques to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data. In the training data, in addition to transforming the image into a tensor, we applied a resize to all the images to have dimensions of 300x300 pixels, so that the many images could be compared. We also set the image to flip randomly with a probability of 50%, and to rotate it by an angle of 45 degrees. In the validation and the test data, only the resize and the conversion to tensor are applied.

- Weighting the labels: The problem of unbalanced labels still remains. To solve it, we decided to assign different weights to the losses of each class. What we want is to give the loss on the dominant class a smaller weight, and vice-versa. For this, we first counted the number of images in which each class was present (P). We then calculate the total number of appearances (TP) in the images by adding the number of appearances for each class. Finally, the weight of each class will be equal to 1 - P/TP, in order to give greater weight to classes that appear less frequently. After printing these values, we verified that the class of speed limit signs was given a lower weight, as we desired.

These processes managed to get our model to correctly predict the cases of the images in which the traffic signs present were not so abundant in the training data.

2 Hyperparameters related to training

Hyperparameter tuning is an essential part of controlling the behaviour of a model. If we don't correctly tune our hyperparameters, our model will not produce optimal results, as they don't minimize the loss function, meaning that our model is more prone to fail.

Because of that, we had to tune in order to achieve better results. We created a configuration file where we could change multiple parameters, being the more relevant the learning rate, which determines the step size at each iteration while moving toward a minimum of a loss function. It is also possible to change the number of epochs that our models will run, the number of workers as well as the batch and image sizes.

We also defined more parameters that can be tuned to change our intermediate model which will be analyzed in-depth further.

- **Number of filters** - Number of filters trained to extract the features;
- **Kernel Size** - Specifies the size of the kernel, which is a filter used to extract the features from the images;
- **Pool Size** - Helps to generalize features extracted by convolutional filters, helping the recognition of features independent of their location in the image;
- **Padding** - Amount of pixels added to an image when it is being processed;
- **Stride** - Filter that modifies the amount of movement over the image or video.

3 Image Classification

3.1 Basic Version

The basic version of our project consisted of using two different well-known convolutional neural network architectures, VGG16 and ResNet and comparing their performance when trained from scratch and using pre-trained weights and fine-tuning. We also assumed that this was a multiclass problem, where each image corresponds to only one class, which is the sign with the biggest area.

3.1.1 VGG16

We first started by using VGG16, which is a convolutional neural network that is 16 layers deep, with its weights already tuned.

We started by first using a pre-trained model which is a saved network that was previously trained on a large dataset, typically on a large-scale image-classification task. The weights of the pre-trained model are built in order to detect 1000 different classes. Since we wanted to adapt the pre-trained model to our specific dataset we used transfer learning to customize this model to detect traffic signs. In this pre-trained model, its weights are frozen during the training of the classifier making it a reliable and fast strategy. We also needed to change the final layer of our network, which gives us the output. We changed it to output only 4 different classes, which correspond to what we are trying to predict: speelimit, trafficlight, stop and crosswalks.

When training the model from scratch, we noticed that each epoch took longer to run and that finishing with the same number of epochs, the pre-trained model gave us better results [5].

3.1.2 ResNet

To compare results with other networks we also used ResNet with pre-trained and customised weights. Even though ResNet is much deeper than VGG16 and VGG19, the model size is actually substantially smaller due to the usage of global average pooling rather than fully-connected layers [6].

ResNet network gave us slightly better results, which we present in the Performance section.

3.2 Intermediate Version

This version consisted of designing and implementing a custom architecture to solve the multiclass problem (only one class per picture). For that, we need to create a new model based on a new neural network whose structure was defined by us. The rest of the process (training and testing phases) remained the same as defined for the basic version.

We chose to define a neural network based on a sequential container, composed of two 2D convolutional layers and a 2D max-pooling layer, each separated by a non-linear activation function ReLU (Rectified Linear Unit), which does not activate all the neurons at the same time. A Dropout layer follows, which randomly zeroes some of the elements of the input tensor with a probability of 0.25 using samples from a Bernoulli distribution. After flattening all dimensions into a tensor using a Flatten module, we apply a linear transformation to the incoming data, generating output samples with a size of 128. To create this Linear module, we needed to know the size of each input sample, which is just the product of the dimensions of the shape of the samples that serve as input for this layer. This shape depends on the number of convolutional and max-pooling layers, so we defined a function responsible for making these calculations and returning the output shape of each transformation.

In the case of the convolutional layers, the output size (O) is the result of the formula O = [(IK+2P)/S]+1, where I is the input volume, K is the kernel size, P is the padding and S is the stride. Then, with an input shape in the format (I, I, C), where C is the number of filters used, the output shape is simply in the format (O, O, C). Notice that the number of filters remains the same. In the case of the max-pooling layers, the output size (O) is the result of the formula I/PS, where PS is the pool size used. The number of filters also does not change.

In this specific case, the input images of the network have a size of 200 (all were resized to a 200x200 dimension), and applying the formula above we notice that each convolutional layer decreases the size by 4 (due to a 5x5 kernel, a default stride of 1 and no padding). Therefore, using 32 filters, the input shape after the first convolutional layer is (196, 196, 32) and after the second one is (192, 192, 32). Followed by a max-pooling module, this shape is reduced to (96, 96, 32) since a pool size of 2 was used, reducing the input size to half. So, the input shape of the Linear module is (96,96,32), resulting in an input size of 96*96*32 = 294 912.

After another ReLU activation function and a new Dropout operation with a probability of 0.5, the data goes through a last Linear layer, where the input size is 128 (has to be the same as the output size of the last linear layer) and the output size is the number of the possible classes, which in our problem is 4.

After defining this architecture, we just need to define the *forward* function, which just passes the input data through all these layers.

3.3 Advanced Version

In this version, we adapted the previous models to solve the original problem, which involves multilabel classification, i.e., each image can now have more than one class present. [2]

For this, we had to define a new *Dataset* class, which should read all the traffic signs present in the image from their annotations. We then generated a binary vector, with four elements (one for each existing class), where each assumes the value 1 if the corresponding label is present in the image and the value 0 otherwise. Finally, we return this vector along with the image, in the format of tensors.

The training phase starts by defining **Adam** as the chosen optimizer in the case of the advanced version. During this phase, we get the predictions made by our model, which correspond to an array of four elements, each with the probability of the corresponding label to be present in the image. Then, we must define a threshold which defines the labels that will be considered to be present in the image based on their probability. Currently, this threshold is defined as 0.5, which means that only the labels predicted with a confidence higher than 50% will be considered to be present. With these predicted labels, we can then compare them with the correct labels in order to evaluate our model.

4 Object Detection

Object detection was built so that we could compare results and visualise the outcomes of the detections using an output image, in this way similar to the first project. Two methods were utilised for this step: the Faster RCNN (two-stage object detection), which corresponds to the basic version, and the YOLO algorithm (one-stage object detection), which relates to the advanced version.

4.1 Basic Version

We chose the Faster R-CNN algorithm over the Fast R-CNN and the R-CNN for the basic version of our implementation. This decision was made because the Faster R-CNN is faster than the other two. Although this is unrelated to our work, the method is quick enough to be employed for real-time object detection. We needed to train our model to recognise our objects in order to implement the Faster R-CNN (traffic lights and stops, speed limit and crosswalk signs). Using the 'roboflow' web interface [3], we constructed a dataset with our images, which we then exported in COCO format. This format is then used in the Faster R-CNN network allowing us to train, validate and test our results. We chose to iterate the algorithm 1500 times during the training process in order to achieve the best potential results. Nonetheless, some labels performed better than others as can be seen in the performance section.

4.2 Advanced Version

YOLO (You Only Look Once) is a real-time object detection system based on neural networks. This algorithm is popular since it is both quick and precise. Before we could execute this technique, we needed to train our custom model to be able to recognise our objects (traffic lights and stops, speed limit and crosswalk signs). This way, we created a dataset using the 'roboflow' web interface [3], which we then exported in YOLOv5 format [8]. After this stage, we had everything in place to begin training and testing our model.

Using YOLO's network, we began training our model in order to generate predictions. We chose 150 epochs as training parameters since more epochs didn't improve our performance, which was already good after ten epochs due to the algorithm's powerful character. It was also chosen to evaluate only values having a confidence level of more than 0.5 to get the best possible results and avoid noisy objects. Before saving the final results to a new folder, the method Intersection over union (IOU) was used with the intention of creating an output box that properly surrounds the objects. If the predicted and the actual bounding boxes are identical, the IOU is 1. This approach removes bounding boxes that aren't the same size as the actual box.

Yolo architecture is more like a fully convolutional neural network passing the image through the network once as opposed to faster RCNN, which does detection on several area proposals and ends up conducting predictions numerous times for distinct regions in an image. Unlike RCNN, it's trained to do classification and bounding box regression at the same time. It should be noted that our machines lacked the necessary processing power to run this model effectively. Therefore, Google Colab came in useful to help us run the programme and come up with some findings and results.

5 Efficacy of methods

During the development of the project, we faced different problems, of which we highlight the most relevant in the following sub-sections.

5.1 Data set not being balanced

The first one is related to the data set not being balanced. The class of the speed limit signs had more than half of the images. If all images were tagged with that class, the model would have an accuracy between 60% and 70%, however this is not what we wanted.

5.2 Multilabel Classification

This was a challenge that we had to handle. We needed to find a way to encode the data to detect multiple classes in the same image. For this, we crated a vector with four elements (one for each existing class), where each assumes the value 1 if the corresponding label is present in the image and the value 0 otherwise.

5.3 Hyperparameter tuning

Besides that, we did not have much time to do hyperparameter tuning. We ended up changing the learning rate, the number of epochs and batch size to test more values to see if these changes had positive or negative changes.

6 Status of proposed methods and fulfillment

During the resolution of this project, the group feels that every proposed strategy and implementation was attempted and tested to attain the best possible results.

For the image implementation section, different classification models based on convolutional neural network architectures were put to the test, and their results were compared.

All three proposed versions were also included in the project. Methods such as VGG and Resnet were tested for the basic version. The intermediate and advanced variants of the problem were also implemented. Multiclass and multilabel solutions were built, and the results were reviewed and compared to the simpler versions.

In the object detection section, both the single staged and two-staged versions were considered, and the results were compared for the corresponding YOLO and Faster R-CNN adopted strategies.

With this in mind, the group believes it has achieved all of the project's objectives and has produced some interesting results.

7 Performance

7.1 Classification

First, to test our program's classification capability, we run each of the implemented versions, keeping the same hyperparameters. More precisely, the program was run with 250 epochs in the case of the multilabel version and 100 in the basic and intermediate versions, a kernel size of 5, a batch size of 16 in the multilabel version and 32 in the basic ones, and we used 32 filters. The images were resized to a fixed dimension of 300x300px.

7.1.1 Basic Version

There was no big difference between the well-known architectures VGG16 and ResNet, as both presented high accuracy (both training and validation), even though the validation loss is not that low, having reached its minimum value around 0.5. The evolution of their accuracy and loss during training can be seen in the figures **3** and **7**. In figures **4**, **5**, **8** and **9** it is also possible to see two example batches of the test data for each model, with the correct labels marked in green and the model predictions marked in red. Figures **6** and **10** present the accuracy and loss values of the test data for each one of these models, where it can be noticed a slight advantage of the ResNet model, although both have achieved low losses and high accuracy. We also would like to emphasize that these values do not represent the performance of our models at all, since both this version and the intermediate only consider the traffic signal with a larger area, which can end up being subjective in cases where the existing signals have approximately the same size, consequently leading the models to predict signs present in the image that however were not chosen as the correct label, considering this prediction as wrong, although we understand the output of the models.

7.1.2 Intermediate Version

Compared to the basic version models, our custom model had slightly worse results, as expected, showing a high loss, with a minimum value close to 1.1, although it still maintained a reasonable accuracy, around 0.83 for the validation data. and 0.75 for training ones. The evolution of these values and some examples of predictions for test batches can be found in the images **11**, **12**, **13**. The accuracy of the test data was decent, around 0.77, although there is a considerable loss of 0.76, as can be seen in figure **14**.

7.1.3 Advanced Version

Regarding the multilabel approach, the results were not so satisfactory, which was expected because it is a more difficult task for the model. We ran the program for each of the implemented models (VGG16, Resnet, Custom Model), and the results can be found respectively in the figures (**15**, **16**, **17**, **18**), (**19**, **20**, **21**, **22**), (**23**, **24**, **25**, **26**).

7.2 Object Detection

For object detection we used two different versions: Faster R-CNN and YOLO. The results obtained in the object detection section were pleasantly good when compared to the previous ones. However, we still noted a great difference in accuracy between Faster R-CNN [7] and YOLO [4] algorithms.

7.2.1 Faster RCNN

Although taking a very long time to train the samples, the results of the test set were not the best. This could be due to a training set underfitting problem, with classes like traffic lights and crosswalks being the most affected. The speed limit class appears to be unaffected, which can be explained by a large number of images in the training dataset with the given class. This demonstrates that the dataset is likewise impacted by this class's overfitting. The figures **27**, **28**, **29**, **30**, **31**, **32**, **33**, **34**, **35**, **36** show some examples of detections. The group was actually expecting much better results from the two-layer object detection, yielding the following average precision results:

Table 1: Test Metrics

Type	Accuracy
crosswalk	0.590
speedlimit	0.820
stop	0.840
trafficlight	0.490

7.2.2 YOLO

Despite the fact that YOLO is significantly faster than Faster R-CNN, the accuracy of the results was not compromised, and the outcomes are substantially better. The conclusion we get from this is that there may be an underfitting problem with specific classes, resulting in lower accuracy, to which Faster R-CNN is more vulnerable than YOLO. The figures **37**, **38**, **39**, **40**, **41**, **42**, **43**, **44**, **45**, **46** show some examples of detections. The results for the YOLO implementation are as follows, first for the validation set, then for the test set:

Table 2: Validation Metrics

Type	Precision	Recall	mAP@.5
all	0.970	0.972	0.985
crosswalk	0.964	0.962	0.979
speedlimit	0.979	1.000	0.991
stop	1.000	0.983	0.995
trafficlight	0.937	0.941	0.976

Table 3: Test Metrics

Type	Precision	Recall	mAP@.5
all	0.896	0.897	0.911
crosswalk	0.863	0.895	0.904
speedlimit	0.975	0.995	0.994
stop	0.926	0.959	0.980
trafficlight	0.819	0.739	0.767

References

[1] Imbalanced Datasets. Last Accessed June 2022. https://discuss.pytorch.org/t/dealing-with-imbalanced-datasets-in-pytorch/22596, .

[2] Multilabel Classification. Last Accessed June 2022. https://debuggercafe.com/multi-label-image-classification-with-pytorch-and-deep-learning/, .

[3] Roboflow. Last Accessed June 2022. https://roboflow.com/, .

[4] YOLOv5. Last Accessed June 2022. https://github.com/ultralytics/yolov5, .

[5] Tranfer Learning for computer vision. Last Accessed June 2022. https://pytorch.org/tutorials/beginner/transfer$_l$earning$_t$utorial.html.

[6] Inception ImageNet: VGGNet, ResNet and Xception with Keras. Last Accessed June 2022. https://pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/.

[7] Faster RCNN information. Last Accessed June 2022. https://blog.roboflow.com/how-to-use-the-detectron2-object-detection-model-zoo/.

[8] How to train YOLOv5 on a custom dataset. Last Accessed June 2022. https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/.

A Results

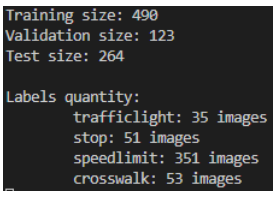

Figure 1: Presences of each label in the training dataset of the Basic and Intermediate Versions

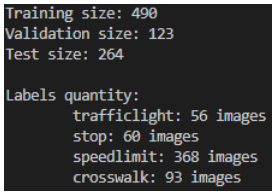

Figure 2: Presences of each label in the training dataset of the Advanced Version

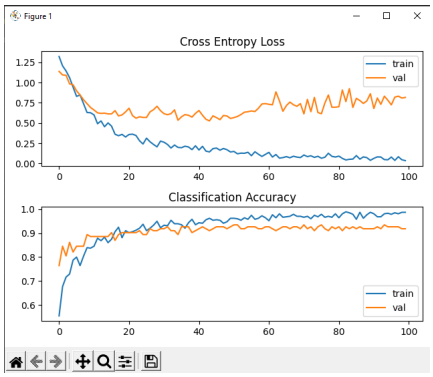

Figure 3: Evolution of loss and accuracy of VGG model in the Basic version

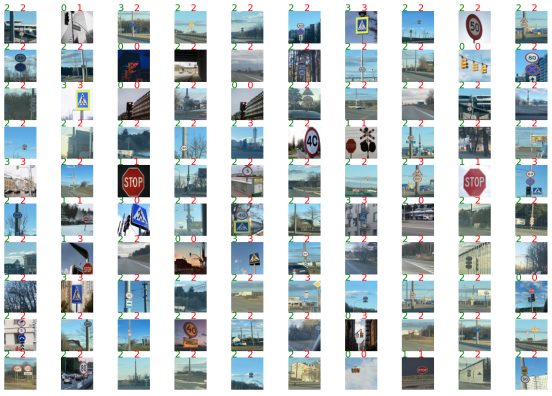

Figure 4: Predictions of the first example test batch by the VGG model in the Basic version

Figure 5: Predictions of the second example test batch by the VGG model in the Basic version

Figure 6: Test Accuracy and Loss of the VGG model in the Basic version

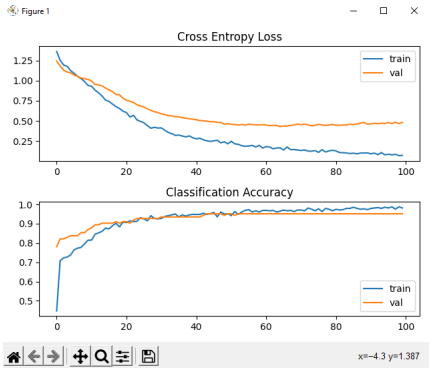

Figure 7: Evolution of loss and accuracy of ResNet model in the Basic version

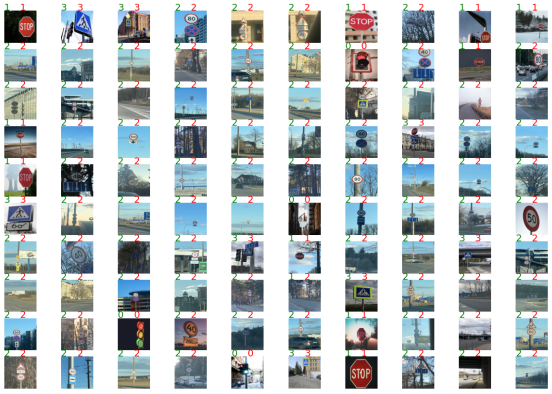

Figure 8: Predictions of the first example test batch by the ResNet model in the Basic version

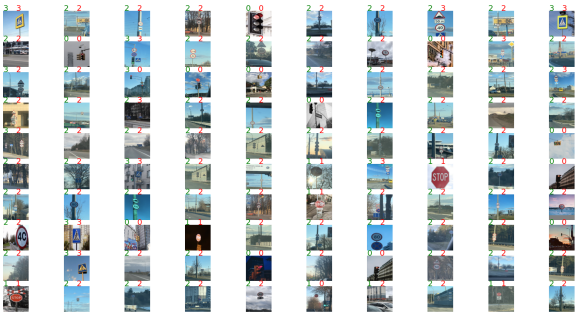

Figure 9: Predictions of the second example test batch by the ResNet model in the Basic version

Figure 10: Test Accuracy and Loss of the ResNet model in the Basic version

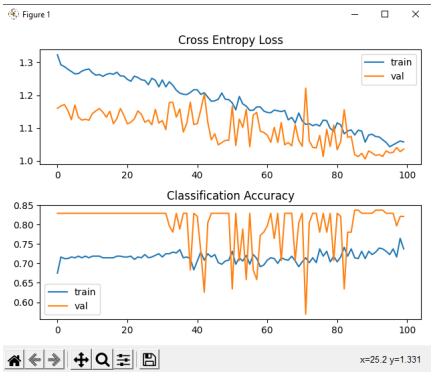

Figure 11: Evolution of loss and accuracy of our custom model in the Intermediate version

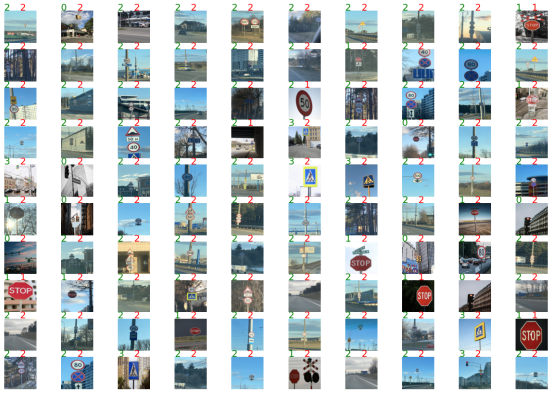

Figure 12: Predictions of the first example test batch by our custom model in the Intermediate version

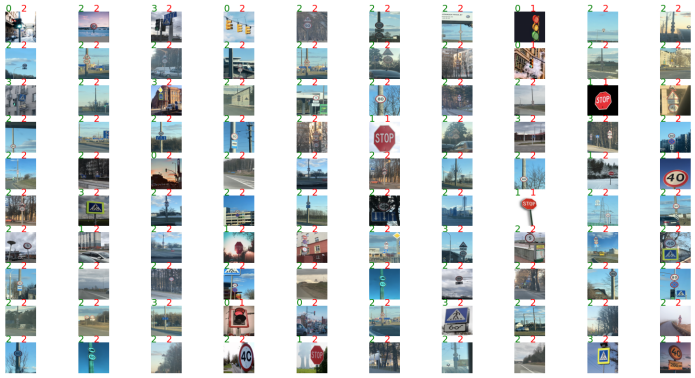

Figure 13: Predictions of the second example test batch by our custom model in the Intermediate version

Figure 14: Test Accuracy and Loss of our custom model in the Intermediate version

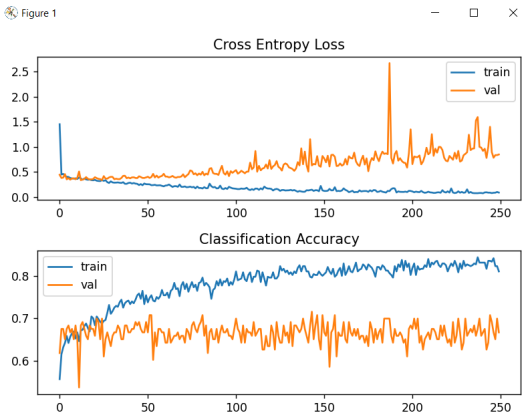

Figure 15: Evolution of loss and accuracy of VGG model in the Advanced (Multilabel) version

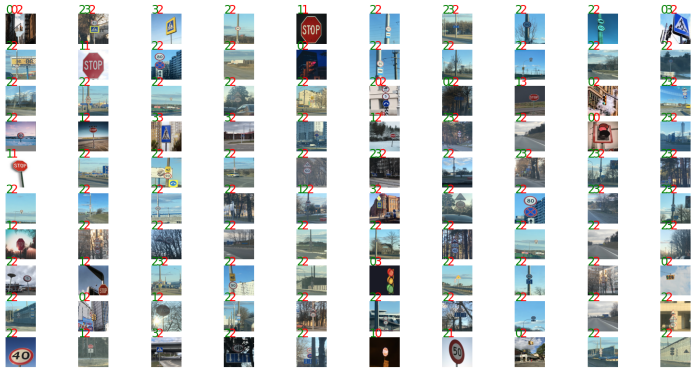

Figure 16: Predictions of the first example test batch by the VGG model in the Advanced (Multilabel) version

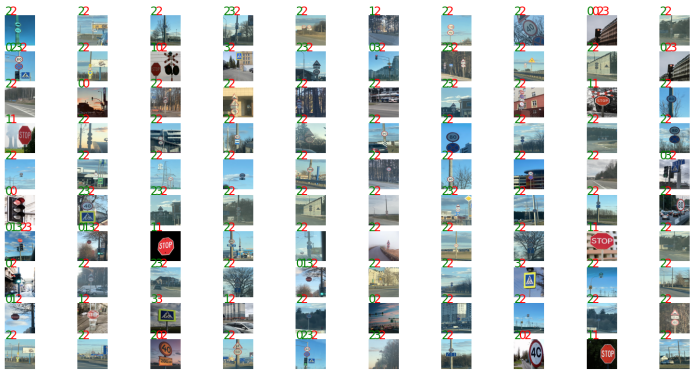

Figure 17: Predictions of the second example test batch by the VGG model in the Advanced (Multilabel) version

Figure 18: Test Accuracy and Loss of the VGG model in the Advanced (Multilabel) version

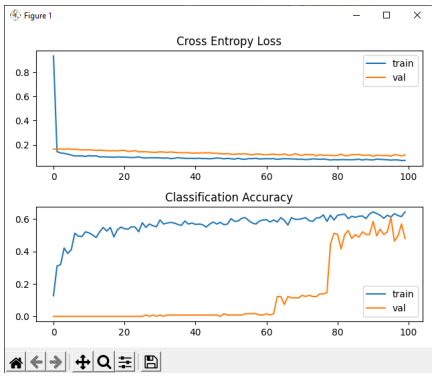

Figure 19: Evolution of loss and accuracy of ResNet model in the Advanced (Multilabel) version

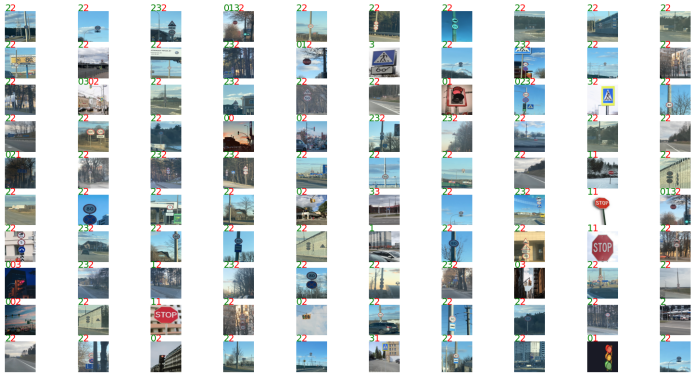

Figure 20: Predictions of the first example test batch by the ResNet model in the Advanced (Multilabel) version

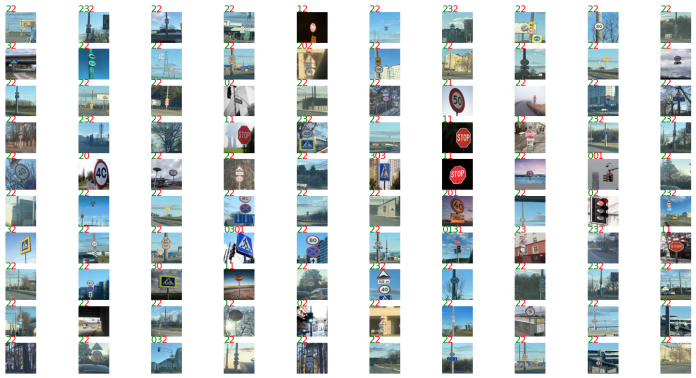

Figure 21: Predictions of the second example test batch by the ResNet model in the Advanced (Multilabel) version

Figure 22: Test Accuracy and Loss of the ResNet model in the Advanced (Multilabel) version

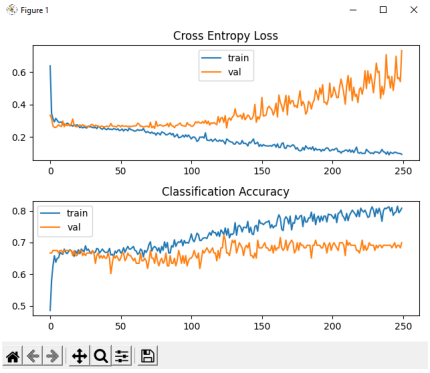

Figure 23: Evolution of loss and accuracy of our custom model in the Advanced (Multilabel) version

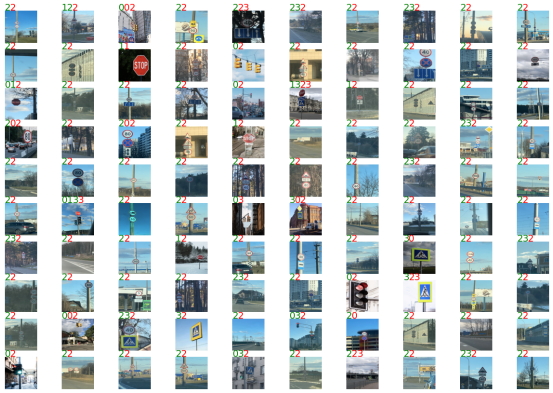

Figure 24: Predictions of the first example test batch by our custom model in the Advanced (Multilabel) version

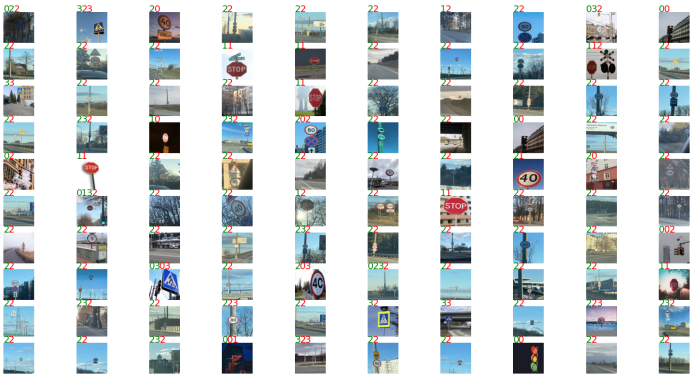

Figure 25: Predictions of the second example test batch by our custom model in the Advanced (Multilabel) version

Figure 26: Test Accuracy and Loss of our custom model in the Advanced (Multilabel) version

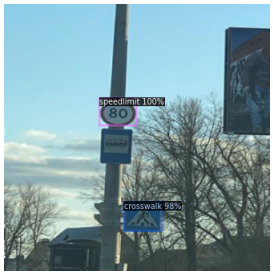

Figure 27: Example 1 of RCNN detection

Figure 28: Example 2 of RCNN detection

Figure 29: Example 3 of RCNN detection

Figure 30: Example 4 of RCNN detection

Figure 31: Example 5 of RCNN detection

Figure 32: Example 6 of RCNN detection

Figure 33: Example 7 of RCNN detection

Figure 34: Example 8 of RCNN detection

Figure 35: Example 9 of RCNN detection

Figure 36: Example 10 of RCNN detection

Figure 37: Example 1 of YOLO detection

Figure 38: Example 2 of YOLO detection

Figure 39: Example 3 of YOLO detection

Figure 40: Example 4 of YOLO detection

Figure 41: Example 5 of YOLO detection

Figure 42: Example 6 of YOLO detection

Figure 43: Example 7 of YOLO detection

Figure 44: Example 8 of YOLO detection

Figure 45: Example 9 of YOLO detection

Figure 46: Example 10 of YOLO detection