Deep Learning for Autonomous Vehicles

Course project - Milestone 1 - Report

Students :

Sylvain Pietropaolo
Yann Martinson
André Gomes

May 10, 2020

Contents

1 Introduction

The project detailed in this report consists in the implementation of an object detector based on machine learning techniques. In the scope of autonomous vehicles, it's mostly about the agents that a car needs to deal with on the road, like pedestrians or all other kind of vehicles that can be encountered. For this purpose, the dataset *EuroCity Personns* [1] (ECP), created specifically for autonomous vehicles, will be used to train an algorithm coming from the state of the art but adapted to this specific case. First, the report will explain the choice of the method used and briefly describe how it works. Then, the specific choices and changes that have been done in order to improve the results will be explained. Finally, the results obtained with the best trained model will be presented.

1.1 Workload

The workload was split as follows (making reference to Table: 2): We all worked on understanding the state of the art and finding a working Google Colab for a similar problem to ours. Then:

Sylvain focused on changing the backbone and the anchors of the different models,
Yann investigated the influence of the optimizer and the learning rate,
André explored how the batch size and dataset size impacted the results.

While Yann compiled the results, Sylvain and André converted the Google Colab used to train the models to functionnal run.py files for both training and testing.

Finally, we all wrote the report together.

2 Selected method description

2.1 Method choice

There is various machine learning methods that can be used to implement an objects detector. They are mainly separated into 2 categories : the 1-stage detectors methods like YOLO or SSD and 2-stages detectors like all the R-CNN methods (simple, fast and faster R-CNN). So the first step is to choose one method among these ones. The ECP dataset website provides a paper [2] where they summarize the results of different algorithms tested on the dataset. The best accuracy obtained, was when using a faster R-CNN algorithm. This method was then chosen to be implemented here to more likely have good results. It should be noted that the implementation of the paper is a variant of faster R-CNN. The method described in this report is thus not the same as the one used in the paper, and one should not expect to found the same accuracy in the present report.

[1] Dataset available on : `https://eurocity-dataset.tudelft.nl/eval/overview/home`
[2] The paper can be found on : `https://eurocity-dataset.tudelft.nl/eval/license/citation`

2.2 Implemented faster R-CNN overview

The implemented solution using faster R-CNN consists in few steps :

1. Building of a suitable dataloader for the ECP dataset
2. Settings of the CNN backbone for features detection
3. Settings of the anchors on the result of the CNN layers to select the proposals
4. Region of interest (ROI) pooling on the proposals
5. Classifier used to find the bonding boxes and their classes

1. Dataloader The dataloader constructed in this method is specifically adapted to the dataset. The data provided by ECP consist in some images and their corresponding labels. A label contains the identities of the objects and the coordinates of the bounding boxes which locate them. These labels are structured by children with one child containing an ECP class. However, one ECP class can contain several classes. As an example the class rider contains the class *pedestrian* for the person on the vehicles and and the class *bicycle* for the vehicles itself. One should know that the ECP dataset contains multiple classes. In order to simplify the training because of the small calculation power offered by the *Google Colab* GPU, only 2 classes are extracted : the pedestrians and the riders. These two were chosen because they are the most represented in the dataset according to the ECP website. In addition, this is also two important classes to distinguish in the autonomous vehicles scope. Finally, the constructed dataloader reads the label corresponding to an image and extract the bounding boxes of the 2 classes *pedestrian* and *rider*. Then everything is regrouped on a target dictionary to be used by the other parts of the method. One should note that the class *rider* is containing the class *pedestrian*, thus for the results a rider should contain 2 boxes : the *rider* box and the *pedestrian* box.

2. Backbone layers The R-CNN methods are two-stages detectors and thus consist in 2 steps. The first one is the region proposal network (RPN). This stage consists in the identification of region of interest where some objects can be found. Then the classifier is fed with these proposals in order to save some computation by avoiding the check of all the initial images several times (to get object with different sizes). In the faster R-CNN method it this done with a convolutional neural network (CNN). The backbone is the architecture of the CNN used to extract some features that will be used to separate the objects from the background. As there is several way to construct a CNN, a suitable architecture for the dataset should be chosen.

3. Anchors settings The output of the CNN layers is a feature map for which each location corresponds to some pixels in the initial image depending on the filters used in the CNN. In order to perform the RPN, loss functions are computed to select if there is an object on a certain location of the image. To avoid the check of the whole image to investigate the presence of objects with several sizes and forms, some boxes are set at each point of the output feature map. Each location of this feature map is corresponding to a certain amount of pixels of the initial image depending on the filters used for the CNN. This boxes of various sizes and orientations are set for each location of the output feature map; this is the anchors. The next step is to compute a loss function to detect if there is an object inside an anchor or

not. The boxes with the best scores are then fed to the next stage of the method as proposal regions.

4. ROI Now that the proposal regions are computed, the next step consists in finding the objects location and their classes. To do so a standard machine learning classifier is used. But the main issue is that the number of parameters for the classifier input is fixed and the number of proposals is depending on the image. The region of interest pooling (ROI) is adapting the number of region of interest to fit in the required number of inputs of the classifier by an usual pooling layer. In addition, as a classical pooling layer the ROI pooling allows to save some computation time.

5. Classifier Finally, the last step consists in a usual classifier based on backpropagation and loss function computation with all the usual hyperparameters like the learning rate, the batch size, the minimization method.

Implemented solution The model used is a Faster R-CNN ResNet-50 FPN. We choose this architecture as it's readily available with Pytorch, with pretrained models (backbone or all) on COCO dataset. The COCO dataset on which it is trained also contains feature of interests for the ECP dataset detection: 'person', 'bicycle', 'car' and 'motorcycle' are for example among the classes contained in COCO. This fully pretrained model has a box AP of 37.0 on the COCO dataset.

2.3 Starting point

The code used in this report is is mainly based on an existing Pytorch tutorial [3] . This tutorial implements a Mask R-CNN ResNet-50 FPN model on the PennFundan dataset. The main changes done relative to this tutorial is to implement a dataloader class for the ECP dataset and to change the model to a Faster R-CNN one.

3 Configuration and parameters tuning

In order to get the best possible results, different configurations have been explored in order to see the impact of some parameters on the model's performances.

3.1 Pre-training

The state of the art allows to use some pre-trained layers by downloading models already trained on large datasets. In this report, the models used are pretrained on the COCO dataset with different levels of pre-training.

First, a fully pre-trained model is taken. It means that all the backbone and the classifier are pre-trained on COCO dataset, which has hundred of thousands images. This has the advantage of having some of the model's layers already set and by applying our dataset, minor changes will be operated on the layers in order to adjust it on the ECP dataset. It thus saves significant computational time. Additionally, having a pretrained model allows

[3]Initial Google colab link : `https://colab.research.google.com/github/pytorch/vision/blob/temp-tutorial/tutorials/torchvision_finetuning_instance_segmentation.ipynb`

to decrease the size of the dataset as it gives a model already able to extract the features without training. Thus, as the size of our dataset might not be big enough because of the small computation power available on *Google Colab*, having a fully pre-trained model might decrease the impact of our small amount of images. The disadvantage of this method is that the model might be less adapted to the present application.

Another way to implement it is to pre-train only the backbone layers. This allow to have a classifier train only on the ECP dataset and may be more adapted to the present application.

Last, we can take the model untrained and apply it to our dataset. The advantage is to have a tailored solution to the set but on the other hand, it is computationally more time consuming. This can play a big role, especially if we have some other changes like the ones below that we want to test. Additionally, as the subset of ECP dataset we are training on is relatively small, we may overfit on the training set and not generalise well later on the testing set.

3.2 Region proposal network

As explained before the first stage of the 2-stages method is to propose some regions where an object can be located through an RPN.

3.2.1 Backbones

The first part of the RPN is the CNN computing the output feature map. As there is a lot of possible architectures to do a CNN, it is interesting to see which one is the most suitable to the present case. That's why various backbones have been investigated. By looking at the state of the art of faster R-CNN one can see that 3 of the the most commonly used backbone network architectures for object detector are *Resnet*, *MobileNet V2* and *VGG 16*. The 3 have been tested in order to select the one that gives the best results. The tests performed in order to see which model could be the best for the present application have all been done with the sames parameters :

Initial learning rate	Optimizer	Training set	Anchors	Pre-training
0.001	SGD	Amsterdam	1.[4]	COCO dataset

In order to compare the performances, the average precision (AP) and average recall (AR) metrics are computed at each epoch. The comparison is mainly done with respect to the first one, the average precision (AP) for intersection over union (IoU) 0.50:0.95 over all areas. As seen on figure 2, the backbone which gives the best results is the *ResNet* architecture with an AP of 20,7% versus 10,6% and 12,8% for *MobileNet V2* and *VGG 16*. One should note that, because of the low calculation power offered by *Google* GPU, the training of this model have been done on a very small dataset (around 1000 images). The results may vary on larger dataset, but it seems to be the best choice with the resources available for this report.

More meaningful results are shown in the appendix on figure 3. The figure 3a shows the bounding boxes for the *ResNet* backbone and the figure 3b for the *MobileNet* one. The first one is better as it has less bounding boxes on the same pedestrian and less classification errors between riders and pedestrians. In addition, there is too many bounding boxes on one pedestrian but this effect might be due to the low number of training images. Finally, for the same number of training images, the *ResNet50* seems to work better.

[4]Details in the section 3.2.2

3.2.2 Anchors

Then, the second parameter of the RPN is the anchors configuration, mainly defined by their sizes, ratio and number per location on the feature map. These modifications impact the proposals used to feed the ROI layer. Different configurations found on the state of the art for object detector are tested to see the possible impact of these parameters.

1. 15 anchors with sizes (32, 64, 128, 256, 512) and ratios (0.5, 1.0, 2.0)
2. 9 anchors with sizes (8, 16,32) and ratios (0.5, 1.0, 2.0)
3. 15 anchors with sizes (8, 16, 32, 64, 128) and ratios (0.5, 1.0, 2.0)

The first test was the initial configuration of the notebook used as a starting point. It was often presenting a lot of boxes onto a single pedestrian or a single rider (see figure 3b). The idea was thus to decrease the number of proposals to see if it decreased the number of boxes by using the configuration 2. One could also note that the overall size of the boxes has also been decreased. Indeed, the configuration of the initial notebook was used for a different dataset. The pedestrian detected was often taking a lot of space on the image and thus the bounding boxes was larger than the ones found for the images of the ECP dataset. The idea was then to decrease the size of the anchors to decrease the size of the proposals boxes. That's also why the configuration 3 was tested. In addition, these anchor configurations have been tested on different RPN backbones to eventually see if there is a dependence between the anchor configuration and the RPN backbone.

As a result, one can see on figure 2 that the modification made on the anchors had a really minor impact on the performances (the AP does not change much). The anchors modification have been done on the *MobilNet* and the *VGG* backbones as it were easier to modify in the initial notebook than on the *ResNet* backbone. Looking at the table 2 one can see that on the *VGG* backbone the change of the anchors have an higher impact on the results (even if it remains small). Thus the decrease of the number of anchors and of their sizes look like having a positive effect on the results (AP from 12,8% to 13,7%). Finally, the results obtained in this section are not very impressive and may lead to the conclusion that the anchor modification is depending on the architecture. Once again these minor changes may be due to the small number of images the models have been trained on, and might be higher on a bigger dataset. Overall, it have been chosen to continue with the initial anchors configuration (1.) for the other models as it lead to the best results.

For more meaningful results, one can have a look on the figures of the section A.2 and A.3. By comparing 3b and 4b the results are worst (more bounding boxes and more classification errors). By looking at 4a it seems that the decrease in size and number of anchors have a positive results on the *VGG16*.

3.3 Optimizer

In deep learning, various available optimization algorithms can be implemented. By choosing two or more different ones, it allows to compare the results, having especially in mind that there is a chance that one leads to an unwanted local minimum, which we try to avoid. Here the two chosen optimizers were SGD and ADAM.

3.4 Batch size

Increasing the batch size often has two direct consequences. First, it requires more memory to load the batch. Second, it decreases the computation time required for one epoch as there will be less steps per epoch. From this, we could conclude that the larger the batch size the better, and that we should take the largest batch possible for the GPU ressources available. However, there seems to be a consensus and studies tend to prove it that increasing to larger batches leads to decrease quality of the model, with less ability to generalize[5]. These results are however true for batches way larger than 8 (often batches between 32 and 512 datapoints are chosen), 8 being the largest we can try due to the GPU limitations of Google Colab. In order to see the impact of this parameter in our model, the following batch sizes are therefore tested:

1. 8 for the training, 4 for the validation
2. 4 for the training, 2 for the validation
3. 1 for the training, 1 for the validation

The results obtained don't give any specific trend to which would be optimal for the ECP dataset, so to reduce the computational time we mainly investigate other parameters with 8 as batch size.

3.5 Learning rate

The learning rate should be chosen with attention, specially with pretrained models. As for any machine learning algorithm, the choice of the learning rate is ambiguous. Too large may lead the algorithm never to converge as it bounces back and forth and too little can take way too long to find an optimal solution.

Here one of the solution was to apply a scheduler to the learning rate. Indeed, the idea behind this is to divide the learning rate by a defined decreasing rate each time after a defined number of epochs. The decrease rate and the number of epochs are then also part of the hyper parameter tuning. In general the goal is to find where the minimum is located with bigger steps and then refine it to converge towards an optimal solution.

3.6 Stopping Criterion

During the training process, a number of epochs is defined. Again, it can't be too low with the risk of under training it but neither too high to prevent the training time to be too long.

Here the approach is to define a maximum number of epochs that can occur without improving the result of the model. We thus record after each epoch whether the model has improved or not. If the number of non improving epochs surpasses the maximum set previously, the training process stops. This allows to set a higher number of epochs but bound in lower if the model is not improving. Usually, this number is set to be between 10 and 30% of the number of defined number of epochs.

[5]The paper can be found on : `https://arxiv.org/abs/1609.04836`

3.7 Dataset size

Most models are trained only on the Amsterdam dataset. It is interesting to note that opposite to what we first believed, increasing the dataset size by taking 2 additional cities didn't increase the quality metrics. Due to memory limitations of *Google Colab*, we limited ourselves to the data from Amsterdam, Barcelona and Basel cities (2966 training and 543 evaluation images). We propose here some ideas of why we get these results, without being able to give confident conclusions.

By increasing the dataset with different cities, we added images with background colors and luminosity quite different from the Amsterdam dataset. It seems that all the images were taken in the same day for a given city. By training only on the Amsterdam dataset, the model was maybe well tuned for such conditions, but unable to generalize well on other datasets. When adding additional cities, the model was maybe forced to generalize better, but being less accurate when it comes to pure metrics comparison. From this we note that comparing models on different datasets, even when it's mostly the same environment (view from a car on the road), is not recommended.

4 Final Model

From the results obtained with different dataset sizes, we decide to give here the best model obtained on training on Amsterdam dataset only. The best model obtained by changing all the other parameters is the following :

Optimizer	Learning rate	Backbone	Pre-training	Batch sizes	Anchors
SGD	0.01	*ResNet50*	Backbone + Classifier	8-4	1.

Table 1: Final model parameters

This configuration leads to a maximum AP around 0.29 depending on the batch sizes for the training and the validation sets. Some images with their bounding boxes are shown in the figure 1 to have a better idea of the accuracy (the sames images with higher sizes are in the appendix B).

(a) (b) (c) (d)

Figure 1: Amsterdam test images with their bounding boxes for the final model

As a result, one can see than the results are much better than for the other models. It has still some classification errors and too many boxes for a single object, but one should keeps in mind that the pictures are the results for a training on a small number of data in comparison with the amount of labeled pictures available on the ECP dataset.

5 Conclusion

Overall, we see that different parameters come into play when it comes to implementing and testing such models. Basing ourselves on open-sourced existing structures, it allowed to dig deeper into how to build the model upon it.

After having modified the structure in order to tailor it to our needs, the goal was to work our way up in a pyramid-like scheme. We did such by testing different configuration but each time having a special focus in mind in ordered to stay structured and see the impact of the focused parameter in order to continue on the best configuration obtained. Small changes outside of the focused parameter could still be applied later, in such a way to maybe find an ideal combination.

As we deal with large datasets and requiring high computational performance, selected choices on backbone, parameters and sizes highly affected computational time and reduced some opportunity windows to increase overall score of the models. This was also hindered sometimes by the GPUs availability.

After around 15 different tests built upon each other, the best combination regarding the resources available could be formed in order to apply it one last time, expecting the best results, which is shown in the final model section. This model reached the best performance based on the scoring criterion that was:

Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100] = 0.29.

Some of the steps that allowed us to achieve the final result were quite challenging. Indeed, we face some troubles to go deep in the understanding of existing codes. Thus, we were

sometimes limited in the change of some parameters. Moreover, the access to the GPU were not an easy task, and we only manage to use the online GPU provided by *Google Colab* but not the one provided by the course. This did restrict the training to a small part of the ECP dataset, which is a big limitation to obtain state of the art results. As an improvement, it should be useful to spend some time to adapt the code to work on GPU with an higher computational power and memory in order enhance the results.

This concludes the tuning of structure and model parameters for the first milestone.

A Configuration and parameters tuning

A.1 Tests

	Focus	Dataset	Backbone					Model			Results	
			Backbone	Pretraining	Batch size train	Batch size valid	Anchors	Optimizer	learning rate	Scheduler	Train loss	Max AP
1	Batch size	Amsterdam	Resnet50	Yes	8	4	Standard	SGD	0.01	6/0.3	0.1097	0.2800
2	Batch size	Amsterdam	Resnet50	Yes	4	2	Standard	SGD	0.01	6/0.3	0.2684	**0.2790**
3	Batch size	Amsterdam	Resnet50	Yes	4	2	Standard	SGD	0.01	6/0.3	0.114	**0.2870**
4	Batch size	Amsterdam	Resnet50	Yes	2	1	Standard	SGD	0.01	6/0.3	0.0866	**0.2830**
5	Backbone	Amsterdam	Resnet50	Backbone only	8	4	Standard	SGD	0.01	6/0.3	0.4639	0.2070
6	Backbone	Amsterdam	MobileNet_v2	Backbone only	8	4	Standard	SGD	0.01	6/0.3	0.2063	0.1060
7	Backbone	Amsterdam	VGG 16	Backbone only	8	4	Standard	SGD	0.01	6/0.3	0.1025	0.1280
8	Anchors	Amsterdam	MobileNet_v2	Backbone only	8	4	8, 16, 32	SGD	0.01	6/0.3	0.0428	0.1000
9	Anchors	Amsterdam	VGG 16	Backbone only	8	4	8, 16, 32	SGD	0.01	6/0.3	0.2088	0.1370
10	Anchors	Amsterdam	MobileNet_v2	Backbone only	8	4	8, 16, 32, 64, 128	SGD	0.01	6/0.3	0.0478	0.1070
11	Optimizer/learning rate	Amsterdam	Resnet50	Yes	8	4	Standard	Adam	0.001	5/0.3	0.2419	0.2078
12	Optimizer/learning rate	Amsterdam	Resnet50	Yes	8	4	Standard	SGD	0.01	3/0.3	0.2595	**0.2840**
13	Optimizer/learning rate	Amsterdam, Basel	Resnet50	Yes	4	2	Standard	Adam	0.005	4/0.5	0.0872	0.2176
14	Dataset size	Amsterdam, Basel, Barcelona	Resnet50	Backbone only	8	4	Standard	SGD	0.001	5/0.3	0.1277	**0.1290**
15	Dataset size	Amsterdam, Basel, Barcelona	Resnet50	Yes	8	4	Standard	SGD	0.001	5/0.3	0.6177	**0.1190**

Figure 2: Tests Results

A.2 Backbone

(a) Prediction on Amsterdam image for test 5 (*ResNet50* backbone)

(b) Prediction on Amsterdam image for test 6 (*MobileNet* backbone).

Figure 3: Predictions for backbone modifications for test 5 and 6.

A.3 Anchors

(a) Prediction on Amsterdam image for test 9.

(b) Prediction on Amsterdam image for test 10.

B Final model