Parking Lot Detection using YOLOv3

MSc Artificial Intelligence - INM705 Coursework

Bohkary, Syed　　García Plaza, Albert

Introduction

The selected domain to perform the coursework is automatic parking slot availability detection. The main objective of the project is to identify the parking lot locations, and whether or not they are occupied given an input image/video. The project has a tremendous potential utility as investigation performed by Conduent Business Services LCC [8] at nine big European cities found that drivers lost around 15 min each day just looking for an available parking spot. Current solutions to this problem are installing ground or ceiling infrared sensors, which are expensive and requires one sensor per parking slot. So, the implementation of parking slot detection using cameras and object recognition algorithms may be a feasible solution, as only one camera should be capturing many different parking slots.

Dataset

To train, tune and test our object detection and classification network, we have selected the public dataset CNRPark, developed by Amato et al. [2].

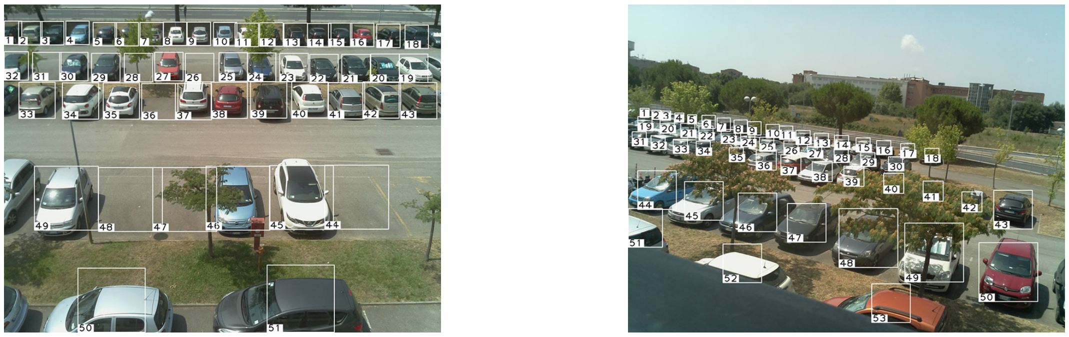


Figure 1: *Two random images from the CNRPark dataset with overprinted bounding boxes.*

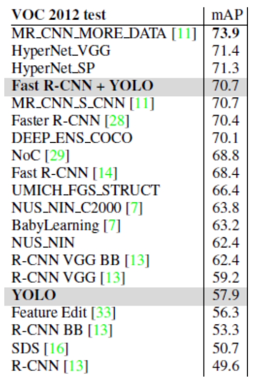
This dataset has a total number of 4,081 images. All images have been recorder using 9 different cameras (pointing different parking slots), during any weather and light conditions. All images have a size of 1000x750 pixels and come with all parking slots labelled with their position and current state (available or not).

Object Detection Network

To perform the parking lot detection, we have chosen the well-known YOLO architecture. This network was developed by Redmon et al. [13] in 2015 and has been improved through different versions. We are going to use the YOLO version 3, as it is the current state-of-the-art stable version of this network [12]. While object detection networks typically take a classifier and evaluate it at different locations and sizes into the image, YOLO follows an entirely different approach. It does only a single forward pass to make its predictions, where a single convolutional network predicts multiple bounding boxes and class probabilities for each box. So, giving an image as input, the output is some bounding boxes surrounding each object, and the probability of these objects belongs to any given class. The network must be trained to detect a specific number of classes, having been trained through a dataset of images and specified bounding boxes as ground truth.

This network has been widely used due to its incredible high speed on the object detection task, as the detection process, it takes only one forward pass rather than being an iterative process like in other approaches. However, this is also its main drawback, as the *only look once* result in smaller accuracy when compared with other widely used networks such as R-CNN or Fast R-CNN [10].

Due to YOLO's high popularity, many variations have been developed from the initially proposed network. WE are going to use the so-called YOLOv3 Tiny, which is a shorter version (less number of convolutional layers) that shows an exciting trade-off between accuracy and computational processing required. While its detection accuracy (on COCO dataset) is 35% worst than the first network, it is almost 5 times faster and more than 7 times lighter.



VOC 2012 test	mAP
MR_CNN_MORE_DATA [11]	**73.9**
HyperNet_VGG	71.4
HyperNet_SP	71.3
Fast R-CNN + YOLO	70.7
MR_CNN_S_CNN [11]	70.7
Faster R-CNN [28]	70.4
DEEP_ENS_COCO	70.1
NoC [29]	68.8
Fast R-CNN [14]	68.4
UMICH_FGS_STRUCT	66.4
NUS_NIN_C2000 [7]	63.8
BabyLearning [7]	63.2
NUS_NIN	62.4
R-CNN VGG BB [13]	62.4
R-CNN VGG [13]	59.2
YOLO	57.9
Feature Edit [33]	56.3
R-CNN BB [13]	53.3
SDS [16]	50.7
R-CNN [13]	49.6

Figure 2: *Accuracy comparison between different networks and mixed architectures [13]*

YOLOv3 Architecture

On this section, we are going to breakdown the YOLO version 3 architecture and the computational path used to detect objects in images.

The network takes advantage of four different types of layers:

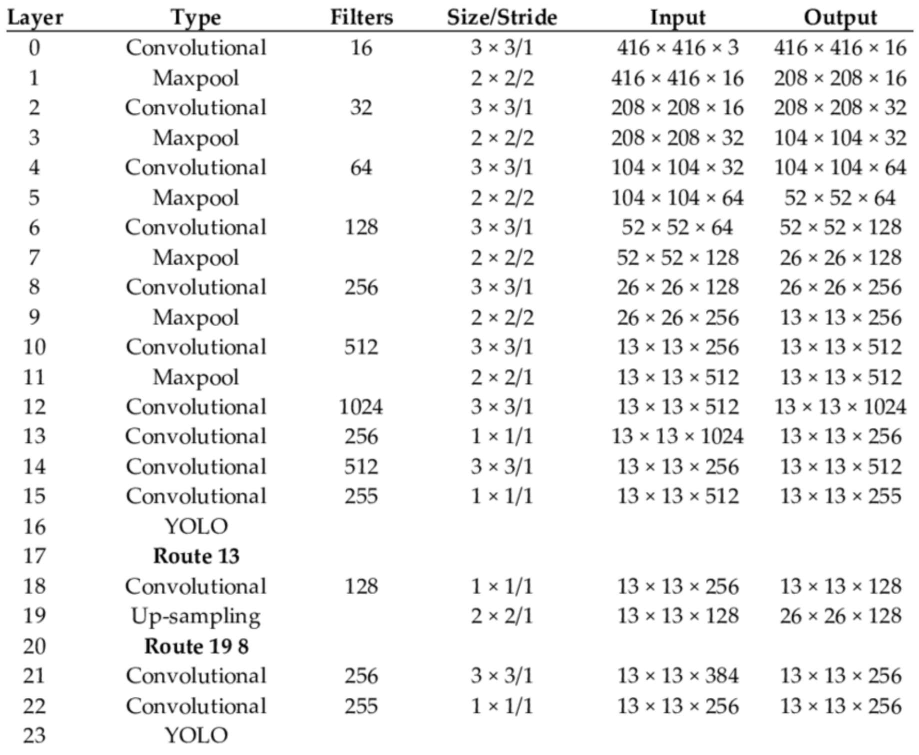
- **Convolutional** layers, all of them with stride 1 (so the image is not downsampled though these layers). The mathematics behind these layers are as follows:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weights(C_{out_j}, k) \star input(N_i, k)$$

  where N_i is the number of images per batch, C_{out_j} is the number of output channels after the convolutional operation, C_{in} is the number of channels of the input, and the $\star$ operator performs the convolutional operation with the specified kernel size and stride.
- **Maxpool** layers between convolutional layers, all of them with size 2x2 and also 2 pixels stride. This operation reads the value of the four cells composing the 2x2 grid filter and returns the maximum value. So, when this layer is performed, the resultant image has its size halved in both dimensions.
- **Route** layers concatenate the output from the current layer where is implemented and other layer's outputs specified by their indices.
- **Upsample** layers perform the regular upsampling process, increasing the size by the specified stride factor.
- **YOLO** layers are the key features of the network. These layers create a grid of specified size along with the

convoluted images and try to detect objects through some proposed anchor bounding boxes.

The whole YOLO v3 tiny layers architecture is as follows:



Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

Figure 3: *YOLO v3 tiny architecture (on Route layers is specified the feeding layer(s)) [6]*

First of all, before feeding the very first convolutional network, the image is re-shaped As has been mentioned above, the main key feature of YOLO approach is the implementation of the so-called YOLO layers. After the first sequence of convolutional layers, when we reach the first YOLO layer out actual input has 512 channels of images with size 13x13 pixels. So, we can explain this as we have generated 13x13 (grid) regions where to search for objects. Then, the image is upsampled until the size has been doubled and the same process is performed -YOLO layer to detect objects- on a grid with double resolution (now it has 26x26 regions). Finally, another upsampling is performed until the image has three times the initial downsampled image size and the YOLO layer will find objects into a three-time bigger grid (52x52). Objects are detected through bounding boxes with n+5 parameters; first, n stands for the number of classes, so the output will be the score (probability) of belonging to each particular class. Second, the objectness score that indicates the probability of detecting an object, no matter its class. The four last parameters are closely related to the predicted bounding box position (x and y location) and shape (width and height) [11].

After the network's forward pass, the **Non-Maximum Suppression** (NMS) algorithm is applied. This algorithm tries to decide if many bounding boxes are detection the same object. That is performed throughout the intersection over union metric (IoU). IF two (or more) bounding boxes are detecting the same objects of the same classes, and their IoU is over a given threshold, we can assume that both boxes are pointing the same objects.

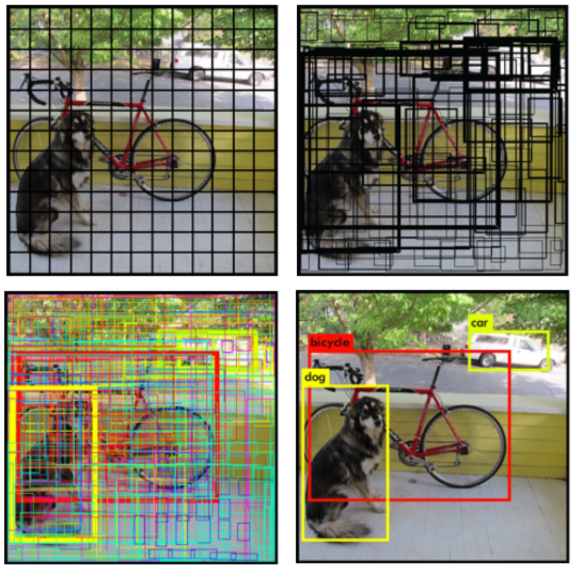


Figure 4: *Sketch of the inference process; first, the image is split into a grid (top-left). Second, then many bounding boxes are proposed (top-right). Third, each bounding box is assigned to one class depending on its scores (bottom-left). Finally, the NMS algorithm merges bounding boxes pointing the same object and only which have scores over a threshold are showed as final result (bottom-right). [Image source https://machinethink.net]*

When training, the backward process will be performed using the **total loss** computed as the sum of different individual losses. These losses are the following:

- **Objectness loss** (L_{obj}): error related to the score of detecting any object on the predicted bounding box.

  $$L_{obj} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} (1 - \hat{C}_i)^2,$$

  where S?? is representing the bi-dimensional square grid on the image (of side-length S), B stands for the number of proposed bounding boxes, and $\hat{C}_i = \sigma(t_o) = IoUPr(object)$, where $Pr(object)$ is the objectness confidence.

- **Class loss** (L_{cls}): error related to the classes assigned to each bounding box.

  $$L_{cls} = \sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p_i(c)})^2,$$

  where $\mathbb{1}_i^{obj}$ is 1 when the bounding box detecting the object of c class has been proposed by the current grid cell being evaluated (i index), 0 otherwise.

- **Bounding Box loss** (L_{box}): this is composed of four other different error parameters; the bounding box x and y position, width and height of the predicted compared with the ones provided on the ground truth.

  $$L_{box} = \sum_{i=0}^{S^2} \sum_{j=0}^{B} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 + (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2].$$

All loss equations showed above follow a $\mathbb{l}^2$ norm, and the width and height loss is into square roots to increase size stability (as bounding boxes will be proposed at three different size ranges). Then, the global loss is the sum of all three individual terms, multiplying each one by a characteristic constant λ_i in order to weight each loss differently:

$$L_{total} = \lambda_{obj} L_{obj} + \lambda_{cls} L_{cls} + \lambda_{box} L_{box}.$$

Methodology

To increase the size of the dataset, being able to perform a better training later, we have augmented the dataset in the following ways: horizontal and vertical image mirroring, and after, we have doubled the dataset adding Gaussian noise to all copied images. So, from the starting number of 4,081 images, we have obtained a total of 24,486 images with their respective ground truth labels (bounding boxes at each image indicating the parking slot and its availability state).

In order to perform the train and testing, we have split the dataset into 3 folds:

1. Block1: Training images and labels with 40% of the dataset (9,794 images).
2. Block2: Second training dataset with also the 40% of the images.
3. Test: Test block containing the remaining 20% of the dataset (4,897 images).

Rather than start the training from zero, we have started with YOLOv3 tiny network weights already trained on *Microsoft COCO dataset* [1]. The weight file comes in the so-called Darknet[1] format. This file is converted into PyTorch's weight format, as all the source code has been coded using PyTorch framework. To write this code, the code written by UltralyticsLLC. [15] has been taken as baseline; cleaning the non-necessary code for our purpose, commenting all lines to follow the workflow easier, and adding the required functionalities to perform our proposed study (e.g. different optimisation algorithms, loss functions or metrics).

Block1 Network Tuning

The objective of this block is choosing which backward pass optimisation algorithm performs better, and tune its hyperparameters (learning rate and l2 regularisation). We are going to try the following algorithms:

- **Stochastic gradient descent** (SGD): this widely used method calculates the derivative from each training data instance and computes the update immediately. This fact makes SGD a high-speed and computationally cheap algorithm. However, due to the frequent updates, the results are very noisy, and this ends having longer training sessions. Also, SGD has a hard time escaping from saddle points. It is fully-controlled by the learning rate (α), where the current gradient is multiplied by this parameter:

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t}.$$

- **Adaptative gradient** (AdaGrad [5]): in several problems the most critical information is presented in the data that is not as frequent but sparse. So, AdaGrad is a modification from the SGD where the learning rate is modified depending if parameters are more or less sparse. The learning rate is divided between the square root of the cumulative sum of current and past squared gradients (v). Also, a fuzz factor (ϵ) is added to avoid divisions over zero:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_{t-1} + v_t + \epsilon}} \frac{\partial L}{\partial w_t},$$

where $v_t = [\frac{\partial L}{\partial w_t}]^2$.

- **Adaptative delta** (AdaDelta [16]): developed as improvement of AdaGrad, this algorithm is focused on the learning rate updating. Delta stands for the difference between the current weights and the newly computed ones. The algorithm removes the use of the learning rate parameter by replacing it by D, the exponential

[1]" Darknet is an open-source neural network framework written in C and CUDA." (Redmon [10])

moving average of the squared deltas. The optimizer behaviour is controlled by two parameters (β_1 and β_2):

$$w_{t+1} = w_t - \frac{\sqrt{D_{t+1} + \epsilon}}{\sqrt{\beta v_{t-1} + (1-\beta)v_t + \epsilon}} \frac{\partial L}{\partial w_t},$$

where $D_{t+1} = \beta[\delta w_{t-1}] + (1-\beta)[\Delta w_t]^2$.

- **Adam** [7]: is an algorithm that tries to leverage the power of adaptive learning rates method to find individual learning rates for each parameter. It takes the advantages of AdaGrad, which handles really well sparse weights, but no so well in non-convex problems. The formulation is as follows:

$$w_{t+1} = w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

where $\hat{m}_t = \frac{(1-\beta_1)\sum_{i=0}^{t} \beta_1^{t-i} \frac{\partial L}{\partial w_t}}{1-\beta_1^t}$, and $\hat{v}_t = \frac{(1-\beta_2)\sum_{i=0}^{t} \beta_2^{t-i} [\frac{\partial L}{\partial w_t}]^2}{1-\beta_2^t}$.

- **AMSGrad** [9]: is currently the state-of-the-art optimizer derived from Adam. It revisits the adaptive learning rate component in Adam and changes it to ensure that the current v is always greater that the one from the previous time step:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{max[\frac{v_{t-1}}{1-\beta_2^{t-1}}; \frac{v_t}{1-\beta_2^t}] + \epsilon}} \frac{m_t}{1-\beta_1^t}.$$

It is important to emphasise that the best-performance optimiser is problem-specific; each problem will work better with one different algorithm and with different hyperparameters setting (usually, each optimiser performs better with different hyperparameters -e.g. there is a different best-learning rate value for each optimiser.

Following the recommendations made on [4], we are going to perform a grid search along the following configurations:

Table 1: *Grid search performed on first block training*

Optimizer	Learning Rate	Weight Decay
SGD	[1, 0.1, 0.01, 0.001, 0.0001, 0.00001]	[0.5, 0.1, 0.05, 0.01]
AdaGrad	[1, 0.1, 0.01, 0.001, 0.0001, 0.00001]	[0.5, 0.1, 0.05, 0.01]
AdaDelta	[1, 0.1, 0.01, 0.001, 0.0001, 0.00001]	[0.5, 0.1, 0.05, 0.01]
Adam	[1, 0.1, 0.01, 0.001, 0.0001, 0.00001]	[0.5, 0.1, 0.05, 0.01]
AMSGrad	[1, 0.1, 0.01, 0.001, 0.0001, 0.00001]	[0.5, 0.1, 0.05, 0.01]

Gamma parameter value has been set to 0.9 for AdaDelta algorithm. Also, has been set $\beta_1 = 0.9$ and $\beta_2 = 0.9999$ on either Adam and AMSGrad optimizers.

Block2 Network Tuning

Intersection over Union (IoU) metric, also known as *Jaccard*'s index, is the most common metric used when comparing the similarity between to arbitrary bounding boxes. IoU encodes the shape properties of the objects under comparison into the region property and then calculates a normalised measure that is focused on their areas -this makes IoU invariant to the scale of the problem. However, there is no strong correlation between minimising the commonly used losses (remember that IoU value is influencing the objectness loss) and improving their IoU values.

It is important to find a metric with that correlation. As an alternative to the traditional IoU value, the following metrics have been developed:

- **Generalised IoU** [14]: defining C as the smallest enclosing object over the recognised object on the image, the GIoU is defined as:
$$GIoU = IoU - \frac{C \backslash A \cup B}{C}.$$
- **Distance IoU** [17]: defining ρ as the Euclidean distance between the centres of the predicted and ground truth bounding boxes, and c as the diagonal length of the minimal square that covers both boxes, the DIoU metric is defined as:
$$DIoU = IoU - \frac{\rho^2}{c^2}.$$
- **Complete IoU** [17]: considering the relation between both bounding boxes aspect ratios, we obtain the CIoU metric:
$$CIoU = IoU - \frac{\rho^2}{c^2} - \alpha v,$$
where $\alpha = \frac{v}{(1-IoU)+v}$ and $v = \frac{4}{\pi^2}[arctan(\frac{width^{(gt)}}{height^{(gt)}}) - arctan(\frac{width^{(predicted)}}{height^{(predicted)}})]^2$.

In order to understand better these four different metrics, we are going to train the second block with all of them, and compare the obtained results.

Results

All results from block1 with mean-average precision (mAp) over 0.5 are showed in the following charts:

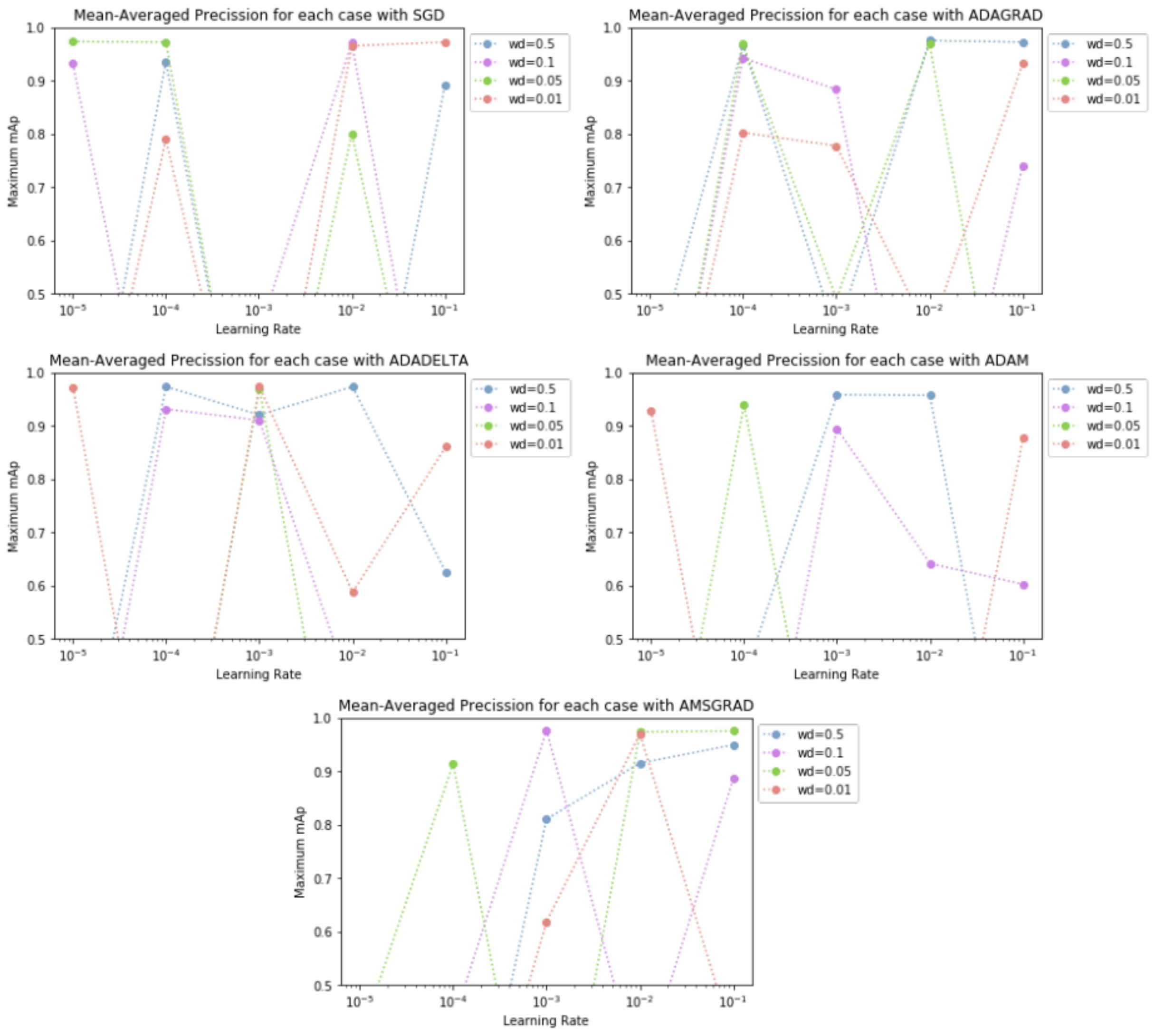


Figure 5: *Results (mAp) of all tried combinations of optimizers algorithm, learning rate and weight decay parameter*

Being all results showed in the following table, where the top-10 mAp have been highlighted:

Table 2: *Results obtained from block1*

LR	WD	SGD	ADAGRAD	ADADELTA	ADAM	AMSGRAD
0.1	0.01	0.891	**0.972**	0.624	0.0125	0.949
0.1	0.05	0.0124	0.74	0.201	0.602	0.886
0.1	0.1	0.00559	0.00765	0.397	0.876	**0.975**
0.1	0.5	**0.972**	0.932	0.86	0	0.394
0.01	0.01	0	**0.975**	**0.973**	0.957	0.915
0.01	0.05	0.971	0.00239	0.391	0.641	0.321
0.01	0.1	0.8	0.968	0.0111	0.197	0.971
0.01	0.5	0.965	0.422	0.587	0	0.97
0.001	0.01	0.00762	0.423	0.92	0.958	0.81
0.001	0.05	0.425	0.883	0.91	0.894	0.971
0.001	0.1	0	0.49	0.97	0.0216	0.0122
0.001	0.5	0.0397	0.777	**0.973**	0	0.618
0.0001	0.01	0.935	0.966	**0.973**	0.423	0
0.0001	0.05	0.0669	0.942	0.931	0	0.429
0.0001	0.1	**0.972**	0.968	0	0.939	0.914
0.0001	0.5	0.789	0.802	0	0	0.00758
0.00001	0.01	0	0.334	0.143	0.141	0.00309
0.00001	0.05	0.932	0.00762	0.00585	0	0
0.00001	0.1	**0.973**	0.00584	0.331	0.928	0.399
0.00001	0.5	0.0058	0.0286	**0.972**	0	0

Finally, those top-10 cases have been evaluated in convergence time (number of steps until convergence has been reached) through the following figure:

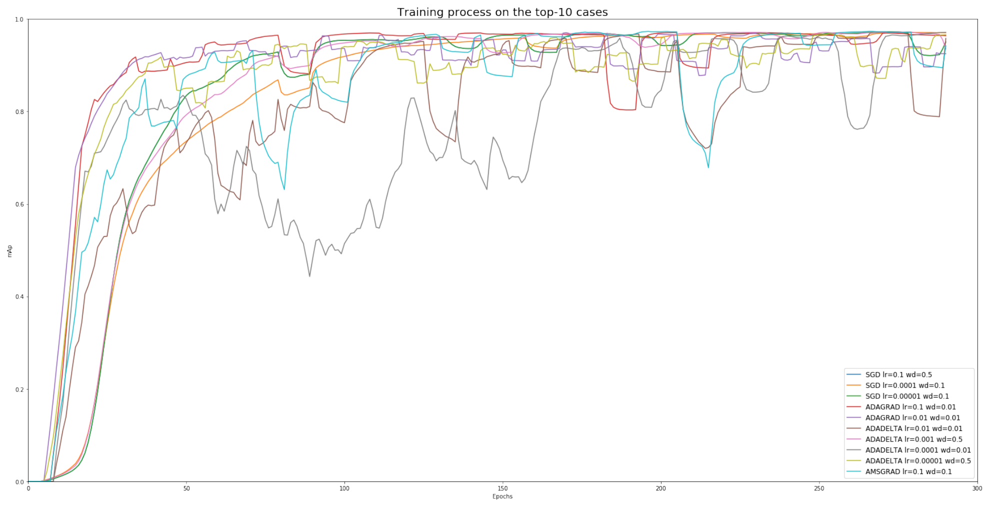


Figure 6: *Results for the top-10 training cases*

On the other hand, Block2 tuning has performed in the following way:

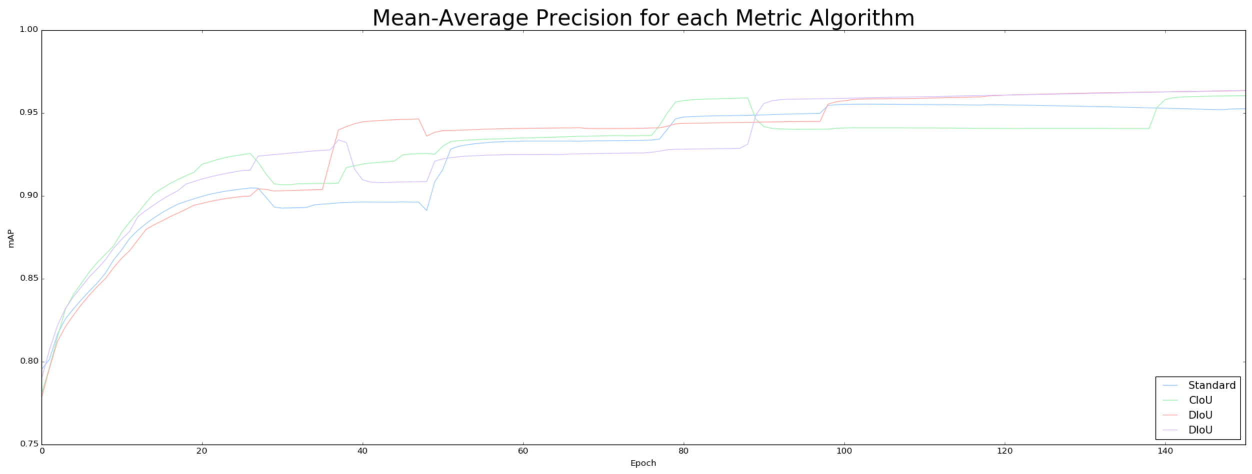


Figure 7: *Results for the block2 training*

Discussion

As overall conclusions about the performance of the trained networks, results obtained by the creators of the dataset [3] have been used to compare the project's results. On their work, they have developed their own network based on the well-known AlexNet, but three times larger.

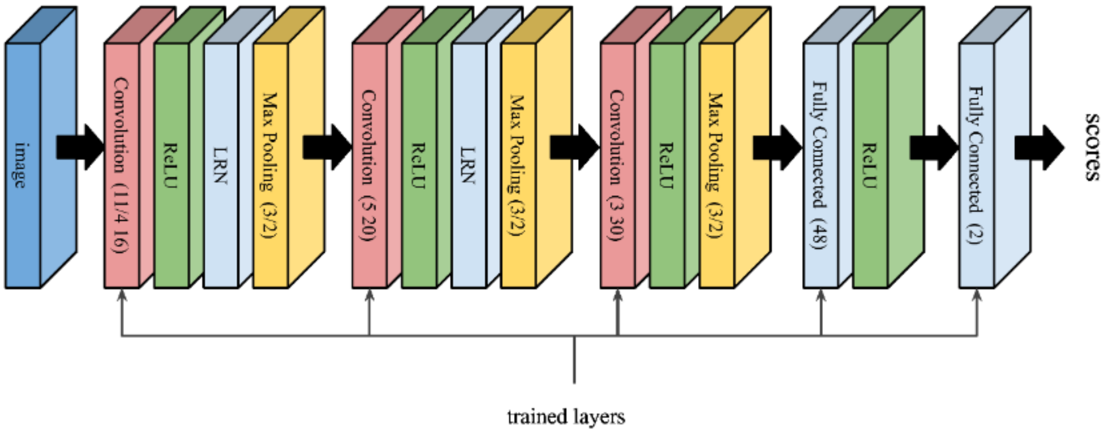


Figure 8: *Network developed and trained by Ultralytics LLC amato2017deep under the same dataset.*

On that network, they have obtained a resulting accuracy on their test set of **95.88%**. However, they do not specify which images have been used to train the network and which ones to test them. So, we cannot make a total comparison between both results.

Our network has reached an accuracy above **97%**, which is slightly better than the one obtained on the aforementioned previous works. So, we can conclude that the trained networks it is fully operable and belong to the state-of-the-art for these considered problems.

From the project's block 1, we have found that all tried optimisers algorithms are suitable for the given problem. All o them have reached, at the end, similar mean accuracy. In general, more prominent weight decay parameters have worked better than with the smaller ones. Also, different optimisers algorithms have converged better with different learning rate parameters.

About the project's block 2, we have no find any substantial difference on the intersection over union metric employed. All four have performed quite similar either in final convergence and in steps required to converge. So, at least for our problem, the classic method may be the best one as is the computationally cheapest.

Some results obtained with our best-performance models over the same dataset are showed in the following figures:

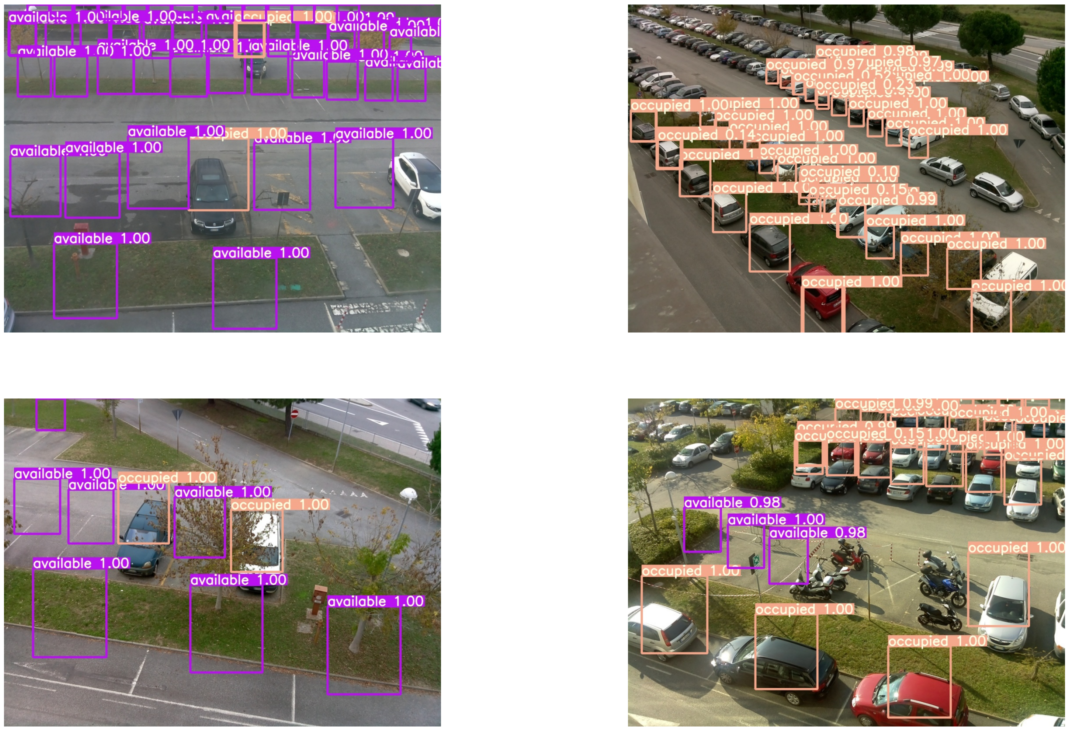


Figure 9: *Parking lot availability detection by the final trained network*

References

[1] Alexey AB. *Darknet*. https://github.com/AlexeyAB/. 2018.

[2] Giuseppe Amato et al. "Car parking occupancy detection using smart camera networks and deep learning". In: *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE. 2016, pp. 1212–1217.

[3] Giuseppe Amato et al. "Deep learning for decentralized parking lot occupancy detection". In: *Expert Systems with Applications* 72 (2017), pp. 327–334.

[4] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *CoRR* abs/1206.5533 (2012). arXiv: 1206.5533. URL: http://arxiv.org/abs/1206.5533.

[5] John C. Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal for Machine Learning Research* 12.6 (2011), pp. 2121–2159. URL: http://dl.acm.org/citation.cfm?id=2021068.

[6] He et al. "TF-YOLO: An Improved Incremental Network for Real-Time Object Detection". In: *Applied Sciences* 9 (Aug. 2019), p. 3225. DOI: 10.3390/app9163225.

[7] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[8] Conduent Business Services LLC. *Keeping our cities moving report -the European urban transportation survey.* 2016.

[9] Tran Thi Phuong and Le Trieu Phong. "On the Convergence Proof of AMSGrad and a New Version". In: *CoRR* abs/1904.03590 (2019). arXiv: 1904.03590. URL: http://arxiv.org/abs/1904.03590.

[10] Joseph Redmon. *YOLO. Real-TIme Object Detection*. Apr. 2020. URL: https://pjreddie.com/darknet/yolo/.

[11] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: *CoRR* abs/1612.08242 (2016). arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242.

[12] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767.

[13] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[14] Seyed Hamid Rezatofighi et al. "Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression". In: *CoRR* abs/1902.09630 (2019). arXiv: 1902.09630. URL: http://arxiv.org/abs/1902.09630.

[15] UltralyticsLLC. *YOLOv3 in PyTorch*. https://github.com/ultralytics/yolov3/. 2019.

[16] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

[17] Zhaohui Zheng et al. "Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression". In: *CoRR* abs/1911.08287 (2019). arXiv: 1911.08287. URL: http://arxiv.org/abs/1911.08287.

Personal Contribution Statement

Both members of the group have developed the presented coursework. We sat down at the very beginning of this second term to choose the topic, and then we search for related information and projects already developed on this dataset and subject. Then, we sat down again to propose our ideas and workflows to complete the assessment until we agreed on that successfully.

The code has been writ en by both of us, working on collaborative cloud platforms, and then we assign to each of us the tasks to simulate and obtain conclusions. Finally, we agreed on the main results and discussion, and we wrote the whole report together.