MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
AUTOMATION DEPARTMENT

A CASE STUDY INTO OBJECT DETECTION USING DEEP LEARNING ALGORITHMS: SSD AND FASTER RCNN

LICENSE THESIS

Autor: **Andreia Maria LAZAREC**

Conducător ştiinţific: **S. L. dr. eng. Cristina Ioana MURESAN**

2022

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
AUTOMATION DEPARTMENT

Vised,

DEAN
Prof. dr. ing. Liviu MICLEA

HEAD OF AUTOMATION DEPARTMENT
Prof. dr. ing. Honoriu VĂLEAN

Autor: **Andreia Maria LAZAREC**

A case study into object detection using deep learning algorithms: SSD and Faster RCNN

1. **The problem:** *Identifying objects while driving using deep learning algorithms.*
2. **Project Content:** *Introduction, Deep learning concepts, Related work, Experimental results, Conclusions, Bibliography.*
3. **Place of documentation:** *Technical University of Cluj-Napoca*
4. **Consultants:** *S. L. dr. eng. Cristina Ioana MURESAN*
5. **Thesis emission date:** *October 1, 2021*
6. **Thesis delivery date:** *July 1, 2022*

Author signature _____

Scientific coordinator's signature _____

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
AUTOMATION DEPARTMENT

Declaraţie pe proprie răspundere privind

autenticitatea lucrării de licenţă

Subsemnatul(a) **Lazarec Andreia Maria**, legitimat(ă) cu CI seria XV nr. 237155 CNP 2990808330204, autorul lucrării *A case study into object detection using deep learning algorithms: SSD and Faster RCNN* elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea Automatica si Informatica Aplicata din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea de Iulie 2022 a anului universitar 2021-2022, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data

1.07.2022

Nume, Prenume

Lazarec Andreia Maria

Semnătura

MINISTRY OF NATIONAL EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
AUTOMATION DEPARTMENT

SYNTHESIS

of the diploma with title

A case study into object detection using deep learning algorithms: SSD and Faster RCNN

Author: **Andreia Maria Lazarec**

Scientific coordinator: **S. L. dr. eng. Cristina Ioana MURESAN**

1. Problem definition: Object detection for self-driving applications.

2. Application domain: Fine tuning of two deep learning algorithms using supervised learning.

3. Obtained results: Good performances for Faster RCNN and ability to detect common objects in the traffic environment.

4. Testing and verification: Computation of mean average precision and loss between for the detectors.

5. Personal contributions: Preprocessing of the used dataset and fine tuning of the networks.

6. Documentations: Scientific papers and books.

Semnătura autorului _____

Semnătura conducătorului ştiinţific _____

Contents

Chapter 1. Introduction

Object detection methods are highly recognized and used in the field of computer vision which further contribute to various tasks and bring solutions to complex matters. One of the applications in which most of its importance and area of development is revealed is ADAS (Advanced Driver-Assistance Systems). The machine and deep learning algorithms along with the fusion between domain-related sensors are significant for assuring safety and fast responses. This not only facilitates quick feedback when taking decisions for avoiding impediments while driving, but also assures comfort for the user. Having in mind the example given above, the output will not only rely on the precision of the existing sensors, but it will strongly depend on the accuracy, agility, and optimization of the algorithms and, eventually, on the trained image detector.

Consequently, there are several directions which have to be taken into consideration when finding the best model for a certain application. The model used and the most suitable approaches for tuning (to get the desired output) can contribute to better results. However, adjusting its hyperparameters or finding methods to reward the network for the predictions determines only one of the aspects that can increase the overall performance. That is why, the great amount of data and its diversity highly contributes to the outcome and helps the model achieve its tasks.

Considering the above-mentioned points, this paper will mainly discuss the comparison between *Faster R-CNN (Faster Region-based Convolutional Neural Networks)* and *SSD (Single Shot Detector)* trained for object detection tasks, using supervised learning as a training method and different tunning algorithms for constant improvement. Having these models trained in such a way, we will investigate the results of each network having as an input automotive-related objects within images (cars, traffic signs and pedestrians). More directions can be eventually developed from this detector which can be tried also without the supporting sensors. One of the future approaches would be to estimate the distance between two cars using the detected object in front.

For accomplishing results that can be eventually integrated into a real-time detector, the dataset must be wide-ranging and large. Therefore, the problems will not appear necessarily when gathering the images – as there are tremendous open-source datasets on the internet – but rather when we will need to label (assign a class to a different object) or annotate (drawing a bounding box to include the object) the data. Augmenting the data can come as a first solution to the related matter which will mirror the objects within the image, turn them to different angles or change the lighting appearances of the overall image. This will certainly help to make the images more diverse without using more external resources.

1.1 Objectives and specifications

In this paper, we will implement two state of the art architectures and use them on a self-driving dataset. The main purpose would be to increase the accuracy of the detector throughout the way by fine tuning its hyperparameters. The dataset, as a very essential

component, would be well thought before starting the training and maybe adjusted by the results we get. This form of training, supervised training, will help us understand how certain hyperparameters in relation to their mathematical explanation work.

The chosen data is an upgraded version of the Udacity dataset for self-driving cars. As it will be observed in chapter 4, this becomes the main aspect that will change and influence our results the most. Therefore, many datasets were taken into consideration, preprocessing more data, not only the one we chose. Nonetheless, modifying the labels or correcting some of the mistakes from the annotations wasn't the case of our study and, because we noticed several mistakes, it can be taken as a future task for increasing the performances of our model.

The tuning of the hyperparameters will be based on several research presented in chapter 3 and also based on the experience that we gained during or after the trainings. We will try to change the main hyperparameters which influence the most the evolution of the training and are significant for our types of detectors. In order to get a deeper understanding of what is happening inside the network, we will provide a some of the essential equations that help us understand how to modify certain hyperparameters. More explanation regarding loss, accuracy and other formulas used for certain deductions during training will be explained in chapter 4.

The final detector resulted from our trainings can be used, along with other components, as an integrated component of ADAS system. These types of application turned to have a high popularity in the automotive industry. While self-driving technology is evolving very rapidly, we have to adapt and find the best solutions that will work in such a context. That is one of the reasons why more than one model was analyzed and also their behavior in different contexts. Therefore, in this paper we will show each step that got us to a certain conclusion and we will adapt the training and its inputs in relation to what we discovered. We will present several methods of fine tuning and what were the results in our case. Future ideas for deploying the application will be also presented as well as future adjustments that could have been made for a higher accuracy.

Chapter 2. Deep learning concepts

2.1. Neural Networks

Artificial Neural Networks (ANN) come as a substitute for our neurons structure due to their similar behavior and ways of computations. As mentioned in [1], the central processing unit (the neuron) will be in charge of performing the mathematical computations, determining one output from a set of inputs. The general structure of a neural network will always contain the input layer, hidden layers (where the actual calculations will be performed) and the output layer (see Fig. 2.1).

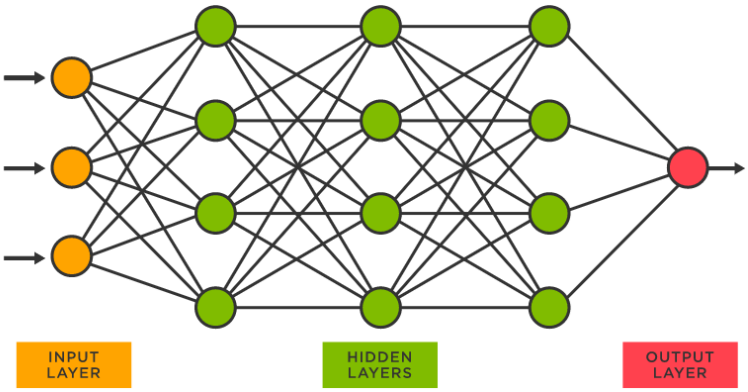

Fig. 2.1 General structure of an ANN [2]

The input layer will contain various samples, in our case, a dataset of images. These inputs will further generate one output gathering the weights from their links and computing the weighted sum of the images and their corresponding weight. The bias (adjusts the output) will be added to the sum and an activation function will be applied over the equation. Hence, taking w as weights, x as inputs and f our activation function for that specific layer, we can express the output of a neuron as follows:

$$f\left(\sum_{i=1}^{n} w_i x_i + bias\right) \tag{2.1}$$

The use of an activation function depends on multiple reasons and the choice of it could change drastically the output of the network regardless of the architecture of the model. Therefore, the need for an activation function comes essentially for giving a nonlinear output. As we can see in *equation 2.1*, before applying the chosen function to the weighted sum, our output simply becomes a linear regression problem. The second use of these functions relies mostly on the type of the activation we used. For example, a sigmoid function converts or normalizes the data very close to 0 for negative inputs and very close to 1 for positive inputs, such that it changes the range of the output and the linearity of the equation. There are many types of activation functions and selecting them for the network depends on

the type of application (statistical purposes, speech processing, computer vision) and on the layer where you apply them to (the sigmoid function is usually applied to the last layer as it can easily classify the output if it is in a certain range).

As mentioned in the *Forward and Backpropagation* section in [1], as we advance in the network from the input to the hidden layers computing each activation function and ending at the output layer, we perform a forward propagation throughout the network. Arriving at the last layer, the algorithm computes the error by subtracting the predicted output form the true one. For providing feedback to the network, the algorithm goes back and changes the weights and biases with more accurate ones – for constantly improving the model – using backward propagation.

In this principle, we firstly need to calculate the cost function by summing up the square of the differences from each output of each class and the real output of each class. For the cost of one training, we get:

$$C = \sum_{i=0}^{n} \left(y_{p_i} - y_{r_i}\right)^2 \tag{2.2}$$

where y_{p_i} is the output predicted by the network and y_{r_i} is the actual output. For contributing on how the network changes its inputs' coefficients, it is necessary to go back to the previous layers – after comparing the outputs – using backpropagation. This algorithm recursively iterates through each layer for adjusting the weights and biases from the negative gradient descent computation. The gradient descent can be represented as a vector composed of the partial derivatives of the cost with respect to the weights and biases of the previous layers. We can compute the gradient descent of the previous layer such that:

$$\nabla C = \begin{matrix} \frac{\partial C}{\partial w_i} \\ \left\lfloor \frac{\partial C}{\partial b_i} \right\rfloor \end{matrix} \tag{2.3}$$

for $i = \overline{0 \ldots no.of\ weights/biases}$.

Highlighting the backpropagation algorithm as written in [3], we conclude that the role of this algorithm is minimizing the cost function. Passing in a reverse direction throughout the network – due to the partial derivative of the cost with respect to the weights/biases – we adjust these coefficients relatively to how much their influence will reduce the cost in a future training. The amount with which these coefficients are reduced is given in *eq. 2.3* accordingly.

Nevertheless, there isn't only one neuron which is deciding upon this gradient, so each assumption on how the coefficients should change depends on more neurons, having more than one training sample, hence, their sum will be performed. For a large dataset, this sum will be very costly when considering the great computational effort already present in each sample. Therefore, one solution would be to separate the data into randomly selected small amounts of data (input images) constructing mini-batches. The gradients will be added after each completion of one set of mini-batches, namely stochastic gradient descent. As stated in

[3], when emphasizing the use of small batches, we voluntarily reach a compromise: taking a single batch as input will give us the best estimate for the gradient, while small batches will fundamentally increase the computational effort and it will provide better results than taking only one sample at a time as input. Other significant aspect in this gradient problem would be to search for that global minimum and to rapidly converge to it. This aspect gained much importance due to the various optimization problems that are suggested. This is a very discussed subject because bad parametrization can lead to instability problems. We could go to this minimum very fast and oscillate very much never reaching it or it could get stuck on a local minimum thinking is the expected global minimum.

Furthermore, the performance of the model will not directly be influenced by how the amount of data was introduced in the inputs of the network (all the data at once or through smaller samples) but rather on how many times the network has seen the data. In correspondence to that, seeing the whole training dataset will mean the network iterated through one epoch. The number of epochs can depend on the size of our batch and on the architecture of the model but, as observing the performances after each epoch, we can eventually change how many times we iterate through data. This number can be modified according to our performances and reduce it especially if we encounter overfitting.

2.1.1 Supervised and unsupervised learning

After we decide on our model's layers and the functions throughout them, we can choose the approach on which the network will be trained: supervised or unsupervised learning. In supervised learning, we give to the network a set of images which are already labeled, and they are included in a class (e.g.: car) so that the network can compare at its output how much it deviated from the true input. Unsupervised learning will let the network decide the labeling based on different algorithms and experience when we have a set of data which we want to use but not label. In this paper, we will cover only supervised learning, but, based on our dataset, some future improvements can be made where unsupervised learning can also be used.

2.2. Used models

Before entering into details regarding the proposed topic, it is essential to make a distinction between classification tasks and object detection tasks. Classification is widely used for numerous image processing purposes and can lead to great results when trying to assign a certain class or label to an image. In this case, the object will occupy the entire image and we only want to know what kind of object we are dealing with and not where it is located within the frame. Therefore, classification is very useful when dealing with signals or medical images where we only need to identify, for example, if a particular disease is observed or not. In these cases, it is also highly important to train with a large amount of data and to spend more time for preprocessing those signals. Object detection adds some features to the classification task, but it is still dependent on it. It is useful when there is needed to know where the object is located within the image. After we identify an object, it is also needed to know what has been detected, so classification will be used and will contribute to the result.

As we wish to detect some traffic-related objects, it is crucial to have knowledge of where certain cars, people or lights are located from our detection spot. Hence, for our project, we will be using object detection as the useful approach in terms of self-driving goal. Nevertheless, even though we will test the final detectors on a real-time environment, it is worth mentioning that image processing solves only a small portion for the self-driving or assistance systems within a car. It is still a useful and required step for the future project and integration and along with several sensors that are assuring a proper communication between them (sensor fusion), could surely contribute to manual driving in a much safer manner. As a stand-alone application, it wouldn't be enough to benefit only from the generated bounding boxes due to a few known aspects. Presumably, the final detector will indeed reveal great performances (small loss and large accuracy) but that will still be restricted to a certain dataset on which it was tuned on. As such, getting it out of its comfort zone could still generate good results if we preprocess the input accordingly, but poorer in comparison. This could generate serious problems if we have even a small amount of false negative detections – not generating a bounding box for a stop sign. Another issue that could be generated is when we try to estimate a specific distance from the detector to the detected object only by using the bounding box or, depending on the situation, by the segmented object. For example, for a cruise control type of application, the distance between cars couldn't be accurately estimated using only the image without any sensor attached.

Both state of the art architectures that are going to be introduced in the next sections will have as their base a convolutional neural network (CNN). This popular algorithm is used for classification purposes and it succeeds in learning better the feature of objects by applying convolution operation during a number of layers. A more detailed explanation will be provided about CNNs in the context of our models. In [4], the importance of convolution is emphasized on how they extract the features from an object and how their simple architecture, by the use of convolution and pooling operation (detailed in 2.3), succeeds in identifying the class of the prediction.

2.2.1. Faster RCNN

One of the networks chosen for training on the selected dataset is Faster RCNN. Overall, this model is known for having high precision and accuracy for object detection tasks due to its mixture of two networks and its algorithms applied throughout the way. We will enter in more details regarding the architecture but, for having a detailed understanding, we will firstly emphasize the evolution of this model from its predecessors: RCNN and Fast RCNN.

RCNN (Region – Convolutional Neural Network)

RCNN is an object detection algorithm that works in two major stages: region proposals and classification (done by a CNN). As such, the way in which it introduced the localization aspect was by proposing, at the first steps of the algorithm, numerous regions of interest (ROIs) from one image. This is done using selective search algorithm which will propose around 2000 regions per image. The selective search algorithm is useful in this case because it reduces all the regions that can be proposed from one image to only 2000 by

eliminating the proposals which could have been included inside other proposals. However, it does this iteratively resulting in large computation time. Due to the variety of objects, the proposals will have various sizes and shapes and each of them will have to be resized when passed as an input to the CNN. The CNN is usually a pretrained network (such as VGG, ResNet, Inception, MobileNet) and will classify each proposal taken from the previous network by applying forward propagation explained in section 2.1. As it can be observed, the algorithm works only if it generates several proposals on which forward propagation is applied resulting in high computational effort and low speed rate for a real-time detection. Its precision can't be doubted, however training such a network with the steps mentioned before will majorly increase the training time that is needed for achieving the desired results. That is one of the reasons this algorithm grew in something which learns and detects faster with better results in accuracy, namely Fast RCNN.

Fast RCNN (Fast Region – Convolutional Neural Network)

To overcome the drawbacks of slow training and detection time, two components that are extremely important in computer vision domain (an improved version of the RCNN algorithm) was developed – Fast RCNN. The main difference that was applied to make it faster and more reliable during computation was the order of steps in which RCNN was performed along with adding a new component in the architecture, RoI (regions of interest) layer. In this model, we won't have 2000 proposals that will enter the convolutional network, but only one image. In fact, the entire image will be transmitted as an input to the CNN determining as an output a feature map. The feature maps are the outputs of CNNs which will result by applying filters and activation functions over the image giving a processed form of the image. As we mentioned in RCNN section, CNN will use the forward propagation algorithm for this input image but in this case, only one time the forward propagation will be requested, not 2000 times. Hence, this comes as a major difference in training time, being reduced significantly. However, the region proposals will still be generated using selective search but on the resulted feature map. These proposals and the feature map will furtherly be passed through the region of interest pooling layer where they will be set to a fixed size. At the end of the network, the actual classification will take place based on the resized regions. So, the changes applied to this network (applying the CNN over the entire image and introducing the RoI pooling layer) have a great effect on the number of frames per second needed for detection as well as the decreasing time for the actual training. Nevertheless, the presented architecture had various improvements that resulted in Faster RCNN algorithm overshadowing Fast RCNN.

Faster RCNN (Faster Region – Convolutional Neural Network)

Faster RCNN comes as a new perspective for solving the problems related to the previous networks that were created. One of the main reasons that determined the need of a new architecture was the time spent on the algorithm in charge of determining 2000 region proposals for an image. Fast RCNN solved the forward propagation issue, but it wasn't enough for having good performances in real-time applications for example, self-driving

cars. So, the idea that was proposed in [5] was to get rid of the selective search algorithm and displace it with a region proposal network (RPN).

The steps that were covered for the RCNN family in the previous examples were kept, same intuition being applied to the model. Therefore, the image will still use a convolutional neural network in which different filters and activation functions are applied to the input image, resulting, as before, in a feature map. However, the feature map will no longer proceed to a selective search method but will be directly taken as an input for the RPN model. The region proposal network is essential in reducing the time spent in the previous cases. As specified in [5], Faster RCNN is maintaining the Fast RCNN network brining, additionally, a new network for generating proposals. So, the RPN will behave as a standard network having its own characteristics that have to be tuned and analyzed. From the feature map, several regions of interest will be extracted having boxes of distinct shapes and sizes. For assuring a filtering process for these regions, they will have assigned a certain confidence score that will tell how much the network believes a certain object is there. In addition, the non-maximum suppression algorithm (explained in 4.3.3) will be applied for reducing the number of final proposals. In many cases, the boxes will cover the same object so it will add more weight on the computation. That is where NMS helps in diminishing this samples. If a certain box will cover the most confident box more than a certain threshold then it will be considered that it belongs to the same object, so it will be removed.

Based on the explanation given above, the next steps for Faster RCNN will be similar to Fast RCNN where the proposals are transferred to a region of interest layer and furtherly passed to the fully connected layer, giving the final prediction for that image. This approach is utterly increasing both accuracy and speed and is a highly used method in object detection due to these advantages. Moreover, Faster RCNN also solved the issue of selective search algorithm which lacked in speed but also in flexibility. When using selective search, the method is a fixed process meaning that the overall network could not learn from the proposed regions and this could lead to various inconsistencies in learning. Even though, Faster RCNN, as the name suggests, is the fastest RCNN from its family, there are still many research papers which try to find even quicker algorithms while maintaining the accuracy of a two-stage detector. As such, another algorithm was proposed and covered in this paper, single shot detector. This one, definitely succeeds in having a better speed than what was created before and it will be covered in the next section.

2.2.2. SSD (Single Shot Detector)

The single shot detector was aimed to be a revolutionary model due to its many advantages. The first strategy that was looked for after the apparition and success of the two-staged detectors was to minimize the time, but maintain the accuracy. That is one of the main aspects that is characteristic for SSD, however, in many cases it has a great accuracy, but it doesn't overshadow Faster RCNN. Its architecture was made for detecting in one stage, reducing the time spent for training and detection as it was in the case of the previous networks.

SSD differs in architecture, being formed by one convolutional network and a single shot detector for its head. The backbone of the model is used for classification, similar to

RCNN architecture, being previously trained on various images and then integrated in the model. As mentioned in [6], the SSD will collect from the feature maps generated by the convolutional network the bounding boxes which have a fixed size. They will have assigned a certain confidence related to how much the detector believes there is a certain object in that box. Afterwards, the non-maximum suppression algorithm will select the relevant boxes and will filter out the ones which are too close to the most confident box. Beside the CNN, that is in charge of returning a feature map of the input image, the SSD is actually formed by 6 feature layers (convolutional layers) that will perform the task of classification as a part of object detection. So, the feature map will be taken through these layers with the purpose of generating boxes of various aspect ratios. Depending on the version of SSD used, the number of generated boxes will vary, but it will be fixed for each object within the image. Similar to RCNN algorithm, a confidence score will be assigned to each box and, based on this score, the first filtering of the boxes will take place reducing them to fewer apparitions. The second filtering will be done by NMS and the final image will contain only one box per object.

From what we have described as a general approach for the single shot detector, we can conclude that there are many similarities to the previous state of the art models. However, one of the main dissimilarities was giving up the RPN network and adding convolutional layers as a substitute. That's one of the reasons why the network can have poorer detection, lacking an entire network dedicated for this task. For weighting this aspect and improving the accuracy, the localization loss was changed from classical cross entropy loss to focal loss, aspect explained in section 4.3.2. For our own experiment, we will focus on both Faster RCNN and SSD models and we will try to analyze their differences both in performance and training.

2.3. Layers

Because we are dealing with architectures that have common backbones networks, CNNs, we can explain and get into details about some of the most important layers that influence the training and how they work in the actual algorithm. Understanding their usage will determine a more efficient training and tuning of hyperparameters. This subchapter will cover the layers used in the convolutional network, but they will also be present as stand-alone layers for other features of the networks. One example would be the convolutional layers that appear outside of the CNN for determining boxes for the SSD. Therefore, the future analysis and reference points for tuning will be also given by the understanding of the algorithm. In the next part, we will present three of the layers that are commonly used in deep learning and have a great impact and importance when applied to an image. They solve numerous problems that were encountered in the past and ease the work that would be passed into training.

2.3.1. Convolutional layer

Image processing is a complex field in which many new algorithms and models are emerging, being constantly adapted for the tasks which appear in this field. From identifying people's emotions to detecting ordinary objects that we encounter on our everyday lives. So,

there is also room for improvement and alterations since there will always be a pressure from the external environment. Classification was one of the approaches that innovate in deep learning area and starting from smaller images (with a fewer number of pixels in them), there was a need to upgrade and use larger images. This is a normal requirement since we want to deploy the model into the real world and try to detect from images that are not processed in this sense before, for easing the human adjustments as much as possible. Consequently, the issue was solved by introducing a convolutional layer in many architectures which made it possible to let the training handle most of the processing of the image.

Convolutional layer, as suggested in its name, will apply the convolution operation over an image by multiplying the kernel of the image with a two-dimensional filter (fig. 2.2). As we want to increase the number of pixels for an image because it simulates the real-world presence of an object, the computational task and extracting of features without convolution becomes nearly impossible. Many pixels will increase the quality of the image but taking each

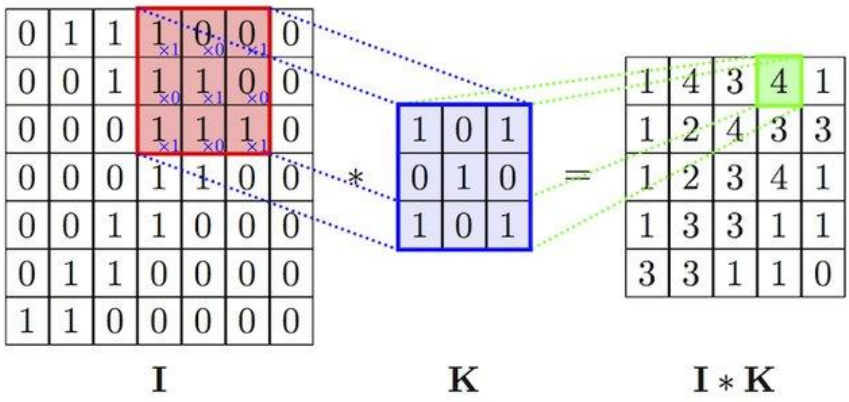

Fig. 2.2 Convolution operation [7]

pixel and comparing them to the others around for extracting a feature can lead to confusion. As it can be observed, the filter, having a fixed size, will be applied over the image for all the available sections. So, the aim for this operation is to identify some patterns which will result in the certain features that we are looking for. Many filters can be used such as filters for blurring or sharpening the image or sobel filter that will find the edges of certain objects.

2.3.2. Pooling layer

The pooling layer comes as a continuation of the convolutional layer and will be applied to smaller regions, as explained in [8]. This algorithm will take a certain input and, based on the size of the resulting matrix and the stride (step applied for arriving at the next pixel) chosen for the image kernel, will result in another matrix as shown on fig. 2.3. Based on the type of pooling used (usually max-pooling), it will extract the highest number from that feature. The intuition behind the resulting matrix is, for a certain feature from the input matrix, there is more probable to have a true characteristic from our detected object. If that resulting number is small, even though we chose the maximum of that set of numbers, it probably means that the feature detected is not present in the object. One of the mentioned

advantages in [8] is that pooling will not be applied on every feature map, as in convolution, but only on the activation maps. This means that it will apply the operation on the already found feature map reducing their size but maintaining their depth.

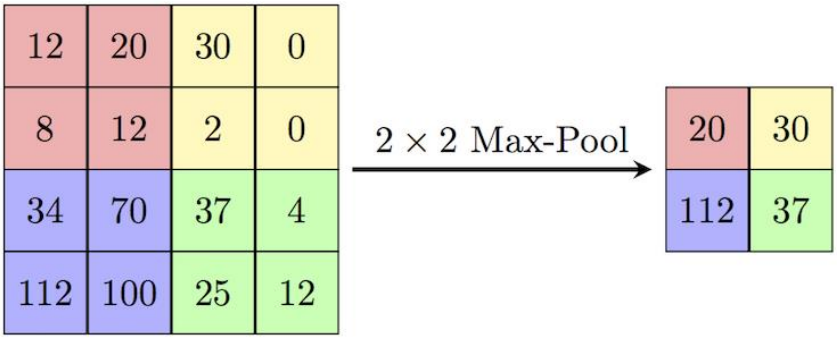

Fig. 2.3 Max-pooling operation [9]

2.3.3. Fully connected layer

When applying the fully connected layer at the end of many hidden layers, the final step – classification – is applied. Therefore, without applying this layer to the last convolutional or max-pooling layer, the network will not be able to make an actual prediction. What is interesting about this layer is that it will behave as a hidden layer, being integrated in the architecture. The input will be nothing else, but the convolutions performed beforehand, while the layer at its right will be the final output layer with the predicted outputs.

In [8], it is specified that the number of fully connected layers is directly dependent on the type of application we want to develop. As for the output layers, the number of neurons will be equal to the number of classes that we set previously in our dataset. Usually, for the final output layer we assign an index that substitutes the name of the class. In the application, we generate a label map script which does this step. At this stage of the network the backpropagation (explained in chapter 2) is performed by giving feedback to the network about the correctness of the prediction. The output layer also uses the loss generated from the true and predicted output and gives this information to network, so that it could improve the future feature maps and generate much better results.

2.4. Activation functions

When choosing a model for a specific artificial intelligence task, we have to understand from the beginning, which kind of data we are dealing with. However, having simple data that can be distributed at the left and right of a line is met in much simpler applications. Our tasks are more demandable and data is often combined and hard to be separated through linearity. As mentioned in chapter 2, equation 2.1 is linear and it needs to become nonlinear for fitting more complex data. That is the main role of activation functions, being used several times throughout the network. There are many activation functions with

different roles which can derive a better solution for a specific task. The most used ones will be presented below.

The sigmoid is a very useful activation to be used inside of a network due to its main advantage, bounding an output to (0,1) interval. However, it is not recommended to use sigmoid for every layer because it reduces the gradient. So, when performing backpropagation, the reduction of the gradient can raise many problems as the model will stop learning. The graphical representation can be seen in fig. 2.4 a. When using sigmoid activation, all neurons will be activated which leads to computational effort. In case of the rectified linear unit function (ReLU), as it can be seen in fig. 2.4 b, only the outputs which are positive will be activated, the rest of them being set to 0. So, this comes as an advantage in terms of computation efficiency. A third very important activation function is softmax. It is mainly used in the last layers, being in charge of determining what is the probability that an object is present.

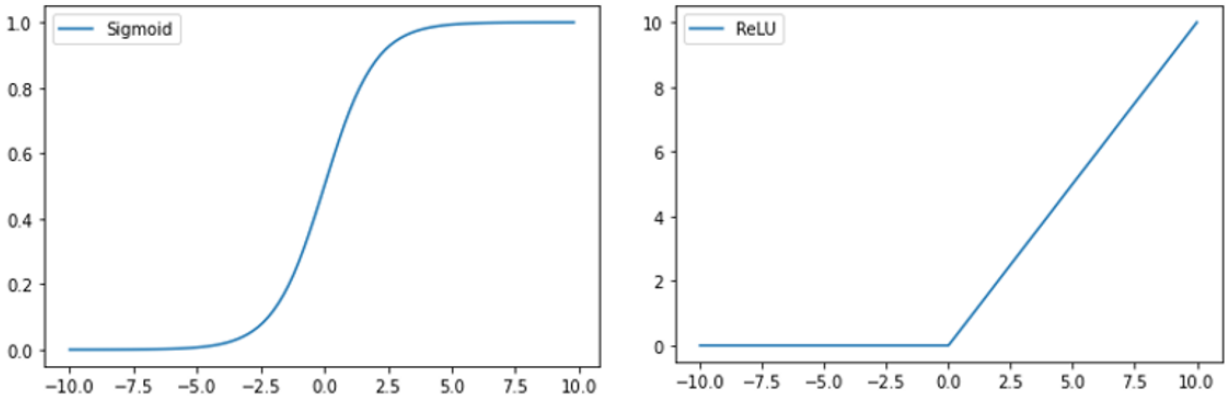

Fig. 2.4 Sigmoid and ReLU activation function

Chapter 3. Related work

Comparing different architectures on the same dataset and with similar changes for hyperparameters is a very used method for tracking the performances. In [10], the author proposed an object detection based only on one class, having the images preprocessed as thermal images. The gray scale thermal images are very useful due to the simplification determined by the lack of colors. This method was chosen in relation to the type of application that will be developed out of the detector, surveillance cameras. They also kept only one class for detection – person – adding simplicity to the initial problem. For this experiment, full trainings (the number of steps needed for a full learning) were performed on the same tuned network but with different backbones. For tuning methods, the anchor ratios and their sizes were changed because the network was trained on COCO dataset and their dataset was formed by aerial images were the object had little number of pixels. For most of the trainings, SSD reached a better percentage for precision, while Faster RCNN had a better recall. A similar approach was given in [11] where the author took also aerial images (colored ones) for determining different kinds of aircrafts pictured from above. However, their aim was to create a framework that could increase the performances on this particular dataset. The preprocessing stage was the essential part from this framework, so they cropped the initial images into patches and then sent them into the SSD network. As for the fine tuning, ratio and sizes of the anchors, they used linear regression over the distribution of width and height and found out the limits for the aspect ratio interval. Along with that, a different algorithm for determining the sizes for the anchors was developed.

An approach which has a different perspective than what was described above is given in [12]. Here, not only the apparition of a simple object is detected (pedestrian), but also their intentions. The authors' motivation started from numerous statistics which revealed that most of the accidents were caused by the mistake of the bikers or pedestrians. In terms of the setup, similar framework was used and the chosen models were Faster RCNN and SSD. The classes that were chosen for this experiment were pedestrians with different movements. The training dataset was quite small requiring only a small number of steps to be fully trained. In this case, Faster RCNN had better performances than SSD. In [13], the same reasoning is applied – accidents are more prevalent due to external factors, in this case, traffic signs. The classes are determined by the variation of 5 traffic signs. Three models were chosen for comparison: YOLOv2, Faster RCNN and SSD. As seen before, Faster RCNN tends to have a visibly better accuracy than SSD, however YOLOv2 outperforms the two models. A more similar application to ours is stated in [14] and it points to different types of cars mainly based on their size. The cars are detected from the front and a rather small dataset was used. In terms of the models that were applied on the dataset, they used YOLO, Faster RCNN and SSD and the same result was received as in the previous case.

We've seen that many of the application are aimed to detect driving-related objects. This is a prevalent subject because of the constant improvements related to safety and

comfort in the automotive industry. Nevertheless, there are many types of applications which come with other perspectives and that are used in computer vision. A face recognition application is described in [15] where not only the faces are detected, but they also counted to how many people are in the image based on the face detection. The idea behind it can be used for different purposes and it can be developed in many areas. The dataset for training had some restrictions from the beginning because they wanted to have mostly people in the image and not many other objects around. However, this simplification of the problem can be seen as the preprocessing part. The Faster RCNN divided in two other projects, both of them concerning the dataset that the CNN was previously trained on. For the convolutional network that was trained on *wider face dataset* they gained good results, while training the backbone on the image-net (person class) generated a major decrease in accuracy. The experiment was ran also for SSD network pretrained on these two datasets and the output was similar. Consequently, this increases the number of methods for tuning a network because more possibilities, as shown in this paper, can be tried to increase performances.

Mask RCNN is an utterly different method for detecting objects, however, in the automotive field, it can have great improvements in detecting and applying the safety methods we want to achieve. The authors of [16] proposed a detection using mask RCNN for observing the damages of a car. The masks are used for providing segmentation to the object by detecting the object's edges. This determines a more demanding type of application and it requires also more computational time. For this reason, they changed the backbone of the model to ResNet50 from ResNet101. As a supplementary preparation of the model before training, they also adjusted the order of the layers of the convolutional network. Transfer learning was used, as in our case, to solve the problem regarding the lack of data.

Chapter 4. Experimental results

This chapter will cover the actual comparison of the algorithms where the checkpoints and analysis of our trainings will be discussed. The results from each relevant iteration will be documented along with the theory and equations that lead us to specific conclusions. The environment which was used for training and validation and the preprocessing of the dataset are going to be also covered as topics for discussion.

4.1. Dataset

One of the decisive steps for the accuracy and good performances of an algorithm in computer vision field is the choice and preprocessing of the dataset. Many applications rely on this component and there is a lack of resources, issue mentioned in many research papers. Selecting the right dataset comes with the analysis of annotations. However, we have to consider also the environment in which the images were taken (lighting and precision) for fitting them on a specific project. For image classification is very important to keep the object within the image frame and to eliminate as much as possible the background because it is not taken as part of the object. The right labeling is also crucial in image processing due to errors that can appear during training if it is not done in the correct way. Also, when working with tremendous amounts of data (needed for generalizing and gaining good results), it is easy to miss or mistaken some labeling and the correction of it could take much effort and resources. Most mistakes can be observed even during or between trainings and due to the large number of hours that are needed to be spent for execution, most of them will remain uncorrected. That is why, finding a dataset with the right setup from the beginning becomes the most significant step for the future training and should be done carefully.

Classification of images requires only one type of labeling, defining which class a certain image is part of and it could be done effortlessly if we go through the dataset and divide into separate folders each class. Finding the right images or creating your own dataset can still determine a challenge in terms of classification and other purposes too. On the other hand, when talking about object detection, the labeling itself becomes more complicated because of the localization aspect. A certain class must be assigned to a bounding box which is formed by 4 coordinates: x_{min}, y_{min}, x_{max}, y_{max}. The first two ones define the left-most point of the box as the last 2 define the right-most point on this rectangular. These coordinates are automatically generated by ground truth editors, but they still need a human annotator to decide classes for each object and to encompass it meticulously. In general, more methods can be used to help with the effort of labeling such as automatic tools or active and supervised learning. However, they are still not as effective as manual labeling and even if they accurately detect the object, some adjustments are needed. For this reason, more steps have to be considered when choosing a dataset for object detection. Labeling objects can determine a real drawback in terms of finding the resources for a project and it is still hard to be found in a vast majority of areas (from medical to many engineering fields). Another aspect that will increase detector's chances to perform well in a defined environment

(detecting from a car) is how we choose the environment in which the images were taken. In our project, the images were taken from a car while driving, monitoring the traffic and other related objects that appeared throughout the way. Therefore, taking cars in different streets and lightings will only help if the dataset is large enough to cover all the scenarios, otherwise it is recommended to use a dataset for your explicit goal.

The main approach for this project was to find a dataset which is related to the traffic or even one that simulates it. In this way, the final detector could be used in self-driving applications and tuned accordingly for the project's objectives. Despite that, cars are indeed a component which cannot be disposed from the application, but there are other objects which can influence the driving as well: stop signs, pedestrians, bikers, traffic lights and many others. So, in order to receive the right objects and their classes for this simulation, we went through various datasets and verified their inputs. The first dataset that was investigated was *BDD100k* which is a very popular dataset for detecting cars, mainly because the quality of images and labels is very accurately done. This dataset has many images from the traffic which are shot in different moments of the day. The major disadvantage that was identified for this dataset in relation to our interest was the weighting of classes. There were a very large number of cars and few labels for objects from other classes. This can undoubtedly influence the training leading to bad results because the network won't learn to generalize appearances of other classes. *Kitti* dataset is also linked to the self-driving field and was taken into consideration for our project. However, the labels were not all set correctly and there were many classes left undefined. For a future annotator, this dataset could be finely tuned, but this wasn't the aim of our research. The third dataset that was tried for this experiment was *Lisa traffic lights* which is an adequately labeled dataset but, as the name suggested, only for the appearance of traffic lights. For this reason, the mentioned dataset was taken into consideration for extracting the images where only traffic lights appeared. However, after going through the images, there were very little images without cars in them and they couldn't be used for our primary scope. The last dataset that was tried which was kept for future executions was Udacity dataset. Its second version was updated with better annotation since the first release and had more than one class. From what was tried before, the Udacity dataset was promising in having the most equilibrate number of labels for each class. In the next section, the stages for preprocessing the selected dataset will be explained.

4.1.1. Data preprocessing

The Udacity self-driving dataset was chosen for training SSD and Faster RCNN models. The number of images which we received in the beginning was around 30000. Therefore, it was nearly impossible to plot or look through the data manually, being a time-consuming process and not a suited approach for preprocessing the data. By skimming over the annotation, we determined each appearance of objects for each class. The dataset was divided in 127900 cars, 7194 trucks, 21491 pedestrians, 3704 bikers and many classes for traffic lights (13673 trafficLightRed, 3482 trafficLightRed-left, 541 trafficLightYellow, 28 trafficLightYellow-left, 10838 trafficLightGreen and 5101 under the name trafficLight). The dataset contains many classes which are of our interest but, at a first look, the problem of class imbalance was present. Hence, the first intuitive thing to be done was to merge the traffic light under one class because we aim only to identify the semaphore and not its color.

If it had been let in this manner, the errors would have been very large due to unbalancing. After merging the classes, the number of traffic lights became 34277. The other two classes, trucks and bikers, were taken out from the dataset because of the small number of occurrences.

Despite this preparation of data, we also decided to keep the images where only the pedestrians and traffic lights emerged for reducing the unnecessary apparitions of cars. This was made by iterating through images where pedestrians and traffic lights appeared while also counting the labels for cars within these images. But even with this modification, the number of cars had still a large value and the final dataset became: 69990 cars, 34277 traffic lights and 21491 pedestrians contained in 9533 images. This can still raise a problem related to unbalancing because of the wider appearances of cars, but we kept this alternative as the final one, being the best from what we have already seen before. The next step, after finding the most optimal solution for these images and their corresponding classes was dividing the images in smaller datasets with different purposes. Therefore, 80% of the images were kept for training, 10% for validation and another 10% kept for the final testing of the detector (or for testing throughout the way). The label map for the classes was created, important step for the final stage of annotation, namely generating the tf-records. These records are responsible for communicating the labels with this specific API, Tensorflow, being referenced as a path in the final configuration file. The tf-records can be considered the final iteration of preprocessing before training, but many other aspects needed to be changed and previous steps were repeated for correcting or adjusting the dataset. As it is an iterative process, even during training we went back for altering the dataset or images and many perspectives were tried. More information regarding new stages of processing images will be discussed in the actual implementation and training.

Before we dive into the training part, there is a significant aspect to be mentioned about a useful preprocessing technique, namely data augmentation. Data augmentation is a powerful method which changes the input images in several ways with the purpose of generalizing the features of an object. Some ways to apply augmentation over the images is by rotating them at different angles, mirroring the object, changing their size or adjusting the color of an image or even luminosity. They are applied at the input of the network along with the existing dataset and they are helping the final detector to better deduce the particularities of an object. In this case, numerous problems can be solved such as overfitting or small datasets. For this project, data augmentation will be implemented at the processing of images using the configuration file, having the ground truth boxes augmented along with the images. However, when using augmentation, we should be converging to the specificity of a dataset. Not all datasets require all types of augmentation as this can reduce accuracy than help increase it. For our example, if we use our detector in traffic, it doesn't have to know how upside-down cars look like and this could reduce judgment than enhance it.

4.2. Setting up the environment

When working with deep learning projects, there are some aspects which are mandatory to be thought of from the early beginning and which can influence the time, effort and results that we generate out of the initial problem. As was specified previously, choosing

the right dataset not only takes a huge amount of time, but decides the final detector's performance. Despite this essential element, the hardware component on which all the algorithms' computation rely on decides the future performances and time spent for tuning and observation. Along with this, the framework used for computation also can determine which architectures can be used for training and the flexibility for selecting any hardware of our choice.

At the beginning of the project, Matlab seemed to be a very reliable source for machine learning or even deep learning approaches. As everything was integrated, the way of manipulating the training, processing of data or different commands was easy to be learnt and understood. The documentation for each network or experiments that were tried in the past were vast and accessible. Nonetheless, the main constrained was the hardware which couldn't be accessed remotely. Some computations were tried and examined on the local CPU but having many images for training took too much time to compute. So, due to this limitation, python was chosen as a more convenient alternative.

The first version of Tensorflow was used as a choice from python frameworks. Tensorflow 1 had many architectures in our interest which included Faster RCNN with different backbones for the convolutional neural network (Inception, ResNet50, ResNet101) and SSD with various backbones (MobileNet, Inception, ResNet50). However, many issues were encountered while training and the flexibility in choosing newer versions of the models was constrained. One of the experiments tried with TF1 will be described in the training section. In consequence, Tensorflow 2 seemed a better choice for implementation and future simulations. The most obvious advantage was the upgrade of pretrained models that were suggested for training as well as their configuration files that could be modified more easily. However, its setup was hard to configure, and the documentation lacked some important aspects that could have eased the initial step. This being one of the reasons why we didn't choose this version from the beginning. Another difference that was noticed from its predecessor was the inability to train and evaluate at the same time, so the validation had to be done in parallel, sometimes even on a different GPU.

The hardware component, being decisive for the next execution, was changed from the local CPU to a remote GPU. Due to the tremendous amount of time spent by training on a CPU and the results that were hardly increasing in performance, a GPU was selected for training in the next iterations. Many machine learning projects are concerned of how to increase the power of computation for the graphical processor unit and this is indeed a resource which can determine the evolution of training from the beginning. What we chose to use in this case was a remote GPU offered by Google Colab. For the sake of comparing the two models, it was a good choice which showed reasonable results eventually. This GPU won't be used for real-time detection, but only for the sake of simulation at the end of the training and for analyzing in the meantime the evolution of the algorithm on the dataset. Some limitations that were encountered while using this resource was the time restriction (bounded to 12h per day). In spite of having these drawbacks, it remained a helpful and easy-to-use resource for this type of investigation, turning into a reliable approach for comparison.

4.3. References for evaluation

Using Tensorflow Object Detection API, we benefited from some integrated metrics which will be used for the analysis and reference of certain performances at each training. The main role of these metrics is to have an idea, even during training or afterwards, where the model is heading and when it stopped increasing the accuracy of detection. For this study, the COCO (Common Objects in Context) detection evaluation metrics are sufficient for returning the right number of performances which help us gather a deeper perspective in what needs to be adapted during training. These metrics were derived from COCO dataset on which our model was previously trained on. They provide a wider range of outputs, each of them being used for trying several approaches of fine tuning and proved to be useful for gaining knowledge about the chosen dataset and network particularities.

These metrics come as a measure of accuracy in different forms of mean average precision (mAP) and average recall (AR) which could be decisive depending on the type of application we want to develop. Precision and recall will be discussed in detail in the next section, and we will cover the formulas needed to compute them based on detection results. However, these metrics are not tightened only to one value, but they cover many more insights as: mAP and AR for small (area of object is smaller than 32^2 pixels), medium (area has a value between 32^2 and 96^2 pixels) and large objects (area has a value greater than 96^2 pixels). Beside these values, it provides how many objects were detected for an intersection over union greater than 0.5 and, for a restrained detection, greater than 0.75. Having this many results can help us understand where the model is, in terms of learning, from various perspectives. In the analysis of Faster RCNN and SSD we will use some of these evaluations and we will tune the hyperparameters based on different sizes for objects both for precision and recall. Nevertheless, as a final statistical view for how well the detector performs, we will refer to the overall mean average precision and we will take it as a reference for the model's final accuracy.

4.3.1. Mean average precision (mAP) and average recall (AR)

As stated in equation *4.1*, when precision is close to 1, it means that our detector had few false positive guesses. A false positive is determined by the unsuccessful attempt of a detector to find an object through a bounding box within an image. Furthermore, if the network did find an object but didn't cover as much from the ground truth box as it should have, then it will still be considered a false positive guess. In equation *4.2*, the concept of recall is introduced. If recall is close to 1, then the number of false negatives is very small. A false negative will not detect an object at all where it should or, in terms of classification, will wrongly label an object as being from another class.

$$precision = \frac{True\ positives}{True\ positives + False\ positives} \tag{4.1}$$

$$recall = \frac{True\ positives}{True\ positives + False\ negatives} \tag{4.2}$$

The COCO evaluation metrics give various cases of mAP as written in their documentation [17]. We can evaluate for different sizes for objects or based on their intersection over union IoU. For AR, we could even evaluate accuracy based on how many objects are detected from one image.

Depending on the computer vision application for which the detector is used, there will be a tendency to see one of the two averages more relevant to one's research. Having in mind the ultimate purpose of our detector: self-driving cars as part of ADAS utilization, it will be necessary to value both. On the one hand, precision will correctly detect a car near us and transmit certain signals to the engine for executing a specific command which has, indeed, a great impact on application performance and safety. On the other hand, having not detected at all an object which should have been there, such as a pedestrian, could lead as well to severe repercussions.

4.3.2. Loss

In this section, the influence of loss in relation to the algorithms used will be firmly established as well as its main characteristics and impact over the training. In this manner, we could also control certain values which directly contribute to a specific loss function. There are several loss functions that could change or influence the path on which the network is learning so it is of great importance to emphasize and understand loss functions' roles. For gathering this kind of intuition around the meaning of loss in machine learning, particularly computer vision tasks, we firstly need to understand the meaning of error.

As explained in *Machine Learning* chapter from [18], loss is defined as how much we lost during a training (utility) and the target is always to minimize this result. Loss will be a variation or error, defined in equation *4.3*. Going deeper into these types of losses and having as a main goal

$$error = y_{real} - y_{predicted} \tag{4.3}$$

to minimize their outputs, we can define, as in [18], the L_1 (absolute value of the error) and L_2 (squared value of the error) losses that are widely used in machine learning literature and have a significant impact on our own analysis too (equation *4.4* and *4.5*).

$$L_1 = \left|y_{real} - y_{predicted}\right| = |error| \tag{4.4}$$

$$L_2 = \left(y_{real} - y_{predicted}\right)^2 = error^2 \tag{4.5}$$

The above formulas take part of the algorithms we will use and such, they will be arranged and varied in diverse kind of losses. Loss becomes one of the most significant metrics that needs to be checked whenever the model is fully trained or during training. In the next sections, the characteristic losses for each model will be explained as well as their effect.

Faster RCNN, as detailed in section *2.2.1*, is a two-stage detector: first proposes regions of interest for the current detection and then classifies them accordingly. In consequence, this detector will benefit also from two stages of losses which definitely boost its way of learning and give the model a proper understanding of its performances in-between two well-defined moments in training. The region proposal network will have two losses: localization and objectness. The former is simply defined as a weighted smooth L_1 loss as in [19] as described in *Object detection* chapter.

$$L_{1_{smooth}} = \begin{cases} \frac{1}{2} \cdot error^2, & |error| \leq \delta \\ |error| - \frac{1}{2} \cdot \delta, & otherwise \end{cases} \tag{4.6}$$

Loss is an essential tool when it comes to ignore or acknowledge outliers. They are isolated values which exceed considerably the normal values from a set of predicted outputs. So, it's of a great importance to choose a method for our type of application in other to gather the right features from loss function. Before explaining the purpose of *eq. 4.6*, MSE and MAE have to be discussed. Mean squared error takes the sum of the squared errors and divides it by the overall value of the dataset, in this way, maximizing the appearance of an outlier. Oppositely, mean absolute error will only take the sum of the absolute value of the errors and divide them over the entire dataset, taking into consideration, in small amounts, the presence of the outlier. Both functions have positive outcomes, but their difference comes in how they sense an abnormal value. In eq. 4.6, after arbitrarily choosing a value for delta, depending above which values we consider the error to be large, we can fluctuate between MSE and MAE. Consequently, we can still find an in-between solution for considering outliers and that is ideally embedded in weighted smooth L1 loss known also as the Huber loss. What especially comes handy about the above equation is the threshold δ which can be modified in such a way that we control which outliers we consider too large for being taken into consideration. If the error is surpassing the selected threshold, we opt for a minimization of this error falling into MAE equation.

Since we covered the significance and properties of localization loss for RPN, we could examine the aspects of classification loss that are present in the first stage of detection. As we previously mentioned in Faster RCNN architecture, we encounter two stages of detection due to the structure of the model and each of these will be separated into two distinct losses: localization and classification. Classification for RPN is represented as a cross-entropy loss,

namely softmax loss. The intuition of the classification for RPN can be tricky as we expect to gain a name for the certain class that has been detected. In reality, this type of classification turns out to be binary and it only tells us if that certain object exists there or not. As the name suggests, we will be using softmax function and apply the cross-entropy loss over it, nothing more than a negative log function. The cross-entropy loss, listed also in [19], will compute the logarithm of the probability of classes. In equation *4.7*, for C number of classes, we sum the log of the probabilities, and we

$$CE = -\sum_{i}^{C} t_i \log\left(f(s)_i\right) \tag{4.7}$$

apply t value which determines the true class taken from the real output dataset. The parameter t will become 0 if the probability of a certain class is not the correct one and 1, otherwise. So, in the end, we will sum up only the negative logarithms for the true classes of the specific output. These probabilities are generated by the softmax function written in 4.8. Softmax function is simply a probability function, having exponential function raised to each probability and divided by their sum of exponentials.

$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \tag{4.8}$$

The box classifier will be determined by Huber loss for localization purposes, already stated in equation 4.6 and the weighted sigmoid loss for classification. The latter will be determined in a similar way like in the case of softmax function. We will apply cross entropy from equation 4.7 over the sigmoid function written in equation 4.9. As we discussed above, the cross-entropy loss will be determined by a sum of log functions which will take as an input certain

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.9}$$

probability. This case will be also similar for sigmoid function which itself is a probability function. Firstly, by specifying different limits for x, it can be concluded that σ will go to 0 when x reaches $-\infty$ and will go to 1 when x reaches ∞. Having this feature of normalizing the prediction between 0 and 1, the sigmoid function will take any chance from the prediction, fitting it into (0, 1) interval. Inside cross-entropy formula, the output for loss will have the same intuition and it will keep only the probabilities from the true classes.

This research will cover two algorithms in which their main contrast is given by the number of stages that are used for detection. Conceptually, Faster RCNN should work slower in terms of frames per second when detecting, but have visibly more accurate results, being a two-stage detector. SSD, on the other hand, has much faster detection time, while its results

are poorer than in the previous case, as a one-stage detector. Nevertheless, SSD was designed with the purpose of increasing in performance concerning both speed and accuracy. Being assembled after Faster RCNN, it had the purpose of overshadowing what has been created beforehand along with having a lighter architecture. For achieving this kind of output, one of the main features that were particularly modified was the loss function.

In terms of SSD's architecture, we will observe an imitation from its predecessor regarding localization loss. Since this algorithm worked quite impressive in past architecture, it was kept for this one as well (Huber loss was used). However, for classification purposes, an improved version of the weighted sigmoid loss was used: sigmoid focal loss. The general formula is based on the classical form of cross-entropy as defined in [20], being adjusted as in equation 4.10. It can be

$$CE_f = -\alpha \sum_{i}^{C} t_i (1 - f(s)_i)^{\gamma} \log(f(s)_i) \tag{4.10}$$

noticed that a new term was added for computing this value and, if we took γ equal to 0, the focal loss would simply become a cross-entropy one. As specified in [20], we can see that γ has a range between 0.5 and 5 and for particular values of confidence (probabilities), the algorithm minimizes the loss so that it won't increase to a much larger number. When using the classical version of cross-entropy and we get a high probability for the predicted classes, we also end up having a small loss. Nevertheless, as loss is still countable, in a case of one-stage detector, this will not be minimized enough. Then, when choosing a value for γ that is different from 0, in case of high probability, the loss will be reduced considerably. When using simple cross-entropy, the certainty of the network is the primary parameter in influencing the resulting probability. As the first graph in [20] suggests, an average probability, for example 0.6, will reach a higher loss for γ equal to 0 than in the case when γ is 2. The meaning behind these values concerns how much we let the model decide on an object even when it is not that confident. So, cross-entropy will tighten the freedom of detection bounding it to a high confidence, while focal loss will let the model detect with a smaller confidence rate. In addition, focal loss helps in better identification for one-stage detectors and has great achievements when dataset is very imbalanced due to the parameter α. This parameter, being less than 1, will influence the classes in different ways, hence becoming a weight that affects uniquely the probabilities.

There is one more function in this category which certainly influences the output of the predictions: regularization loss. Regularization loss will be added to the total loss for SSD network (for Faster RCNN we keep this loss 0) and is the loss generated by the regularization function. The role of this function is to add a penalty to the loss to prevent overfitting and, as a result, it can have a large value in the early stages of training, but it will normalize during the process.

In this chapter, losses that directly concern the used models were covered. Both for Faster RCNN and SSD, this parameter has a huge impact to the flow of training and especially because it is based on various probabilities of guessing the right location and class. As

detection blends two concepts for defining a final accuracy, loss has an even higher rate of significance for the output. The goal of explaining the losses was to have a wider knowledge and to understand which hyperparameters can be modified for getting the most out of the training.

4.3.3. Non-maximum suppression (NMS)

Detectors that we implemented during this project will never propose only one box for an object because they are influenced by another parameter – confidence. Therefore, many bounding boxes proposals will result from a particular detection. Non-maximum suppression (NMS) is one of the algorithms which assists the network when choosing from many proposals. For one object, in the classification stage, the network will propose more bounding boxes based on the confidence score. Then, the main role of NMS will be to filter these bounding boxes until it deduces one that has the highest confidence score.

The NMS will use intersection over union (IoU) for eliminating bounding boxes which could be very close to the real one, filtering them out. It will choose the highest confidence score for a proposed bounding box and will compare the other ones with the selected box by applying intersection over union as described in eq. 4.11. As the equation shows, this intersection is no more

$$IoU = \frac{area\ of\ overlap}{total\ area} \tag{4.11}$$

than a ratio between two areas, the result being bounded to (0, 1) interval. The area of overlap is the area where the selected proposal with the highest confidence and one proposal from the list of proposals intersect. The total area is the reunion of these two bounding boxes, which is why the ratio's outcome will be less than 1.

However, this approach is based on a threshold that will eliminate the other proposed boxes if they hit a higher value than the set threshold. As emphasized in [21], this method is great whenever we have many proposal surrounding the same object so, if eliminated, we will increase the performance of the output because they will be considered false positives. On the other hand, having objects gathered and assigned to the same class (one car in front of the other) we could get rid of a true positive box. So, if the first supposition is our case of detection, then the precision will definitely increase, but for the second supposition we will fail to detect objects that appear in the proximity of other objects, leading to a decrease in recall. Hence, the final purpose when using NMS technique is to overcome the tradeoff between precision and recall.

In [21] is specified how detection leads to a similar hypothesis as in classification. As it was mentioned in the previous section, classification reduces to a probability problem and in detection, as the confidence influences the choice of the real box proposal, it also becomes a probability. So, choosing a threshold still has to work for getting rid of the multiple boxes assigned to the same object, but not influence the near boxes from a near object. What can be considered as a hyperparameter in our configuration is the intersection over union threshold along with a score threshold. The latter will filter the confidence for classification

which is indeed very useful for the raised problem because we don't want our detector to be very confident about every proposal it generates. The values selected for our trainings along with their reasoning will be discussed in 4.4.2 section.

4.4. Results of training, validation and testing

The first experiments that will be covered in this paper will concern the differences and abilities of two models: Faster RCNN and SSD. In order to have a stable and appropriate comparison, we will keep the stages of training and gaining results as similar as possible. Regarding that, we will be sure to compare the same modifications that we applied on the images and change same hyperparameters if needed. As the architecture and the approaches of the models are vastly different, we will stick to similar modifications within this mentioned aspect.

Active learning can be added as a future step applied to this dataset, in which we want to acknowledge if we could have used less annotated images which covered the final task as achieving a similar precision and a similar loss. The application of this method is to select in a more appropriate way the order of the input images as they enter the network to help it learn more with less data. Also, another important quality of active learning is letting the detector – found in distinct stages of its learning – to annotate itself the images and decide the labeling of the objects, based on certainty and uncertainty metrics.

As mentioned before, the aim of these experiments is to fine tune two of the multi-scale detectors and eventually use them in other trainings or to perform real-time detection documenting their performances. Therefore, this process can only work iteratively, and should have many tries before starting the actual tuning of the network. As such, we checked numerous versions of our datasets and pipelines (configuration files) to gain an intuition of how the API works merged altogether with the dataset and the chosen architectures. In the next section, we will cover only the attempts which ultimately made us change specific hyperparameters or gave us a better understanding of where the work is heading. Hence, we will go through particular checkpoints of the model which were part of many trainings.

4.4.1. Choosing the architecture

After completing the preprocessing of the images, several datasets were created to observe the changes of the detectors for more than one use case. For the first iterations we will only use 2 classes from the dataset (car and pedestrian) and improve some features only based on those, eventually adding the third class (traffic light).

One of the main experiments tried was training using Tensorflow 1 API. For this, we used the first version of the dataset having the classes and labels as explained in the section before. A Faster RCNN algorithm was used having as a backbone in its architecture Inception net 2nd version. Because it was a trial test that made us understand how the API and algorithm works, the default parameters from the configuration file had been kept. The learning rate was constant for the whole training and only 2 classes were selected for this try: cars (34934) and pedestrians (21491). We trained the network for 10000 and the main, visible problem that appeared was the increasing of loss on the validation dataset captured in fig. 4.1. This was considered an unexpected behavior, especially because the model

learned some basic features about the objects. For what has been observed, the detector failed mostly to identify the smaller class (pedestrians) and was assigning many boxes to objects which weren't of our interest classifying them as cars. Nonetheless, what has been interesting to be noticed was the detector assigning car labels to trucks (fig. 4.2), class that has been subtracted from our training. Therefore, we decided to reintroduce this class because it will cause problems as the detection will be considered a false positive and we don't consider it to be wrong in its detection. For not destabilizing the classes by making them more unstable, we changed the truck class to be labeled as car.

Fig. 4.1 Increasing loss for the first iteration

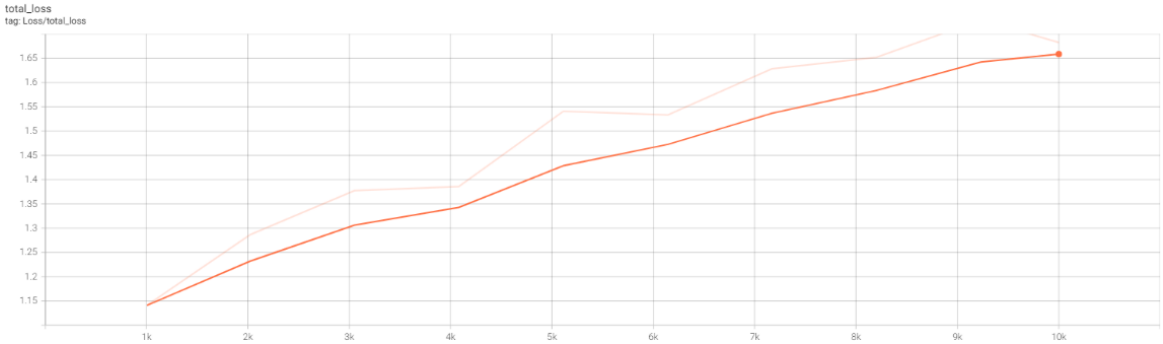

Fig. 4.2 Detector classifying truck as car

Due to the restricted models and high uncertainties in training, we decided to move the next trainings to Tensorflow 2 API. As a first step, a much smaller version of our dataset was used (50 training images and 11 validation images) which were filled in only one of the models for verifying our setup: framework, tf-records, labeling and results of the model. This was useful for preventing some errors which are not directly bonded to our training. What is expected from this first try is to encounter an overfitted model. This, in essence, is not an aim for the lastly trained detector, but with a small number of images and many training steps, we should notice that the chosen model learned the proposed dataset and is performing well on the images that it has seen before. At the validation stage, it is clear that with so little examples it cannot have the capacity for generalizing.

Faster RCNN was chosen for confirming the correctness of dataset and setup, having a ResNet50 backbone in its architecture. The model was trained on the default parameters and hyperparameters, having changed only the pixels for resizing which were initially 640x640. As our images are smaller and their maximization had poorer results in other observations, we changed it to 512x512. The metric used for verifying the overfitted results was given by the total loss functions computed as the sum of classification, localization and regularization loss. In fig. 4.3, training and validation values can be compared after 700 steps where training had an acceptable loss value (around 0.1). The value of the evaluation loss is much higher (around 0.9), so it detected much worse on a new dataset. We could have continued with more steps as we aim to reach a smaller loss, but comparing the loss from validation, the overfitting was already present. Loss lets us visualize the performance of our network throughout many steps in time. As it was mentioned in section *4.3.2*, a small total loss shows the precision of the proposed bounding boxes, namely, how close they are to the real ones and how it succeeded in classifying the object to its real class. Noticing the decrease in loss for our training dataset, it can be concluded that the model performs well on the images that is has seen before.

From the knowledge gained from this experiment, we can conclude that the overfitting was present in this smaller version of the dataset. Therefore, it is expected to have a properly installed framework and well-generated tf-records. Furtherly, we will train the model on the entire dataset, maintaining the default parameters and hyperparameters of the networks.

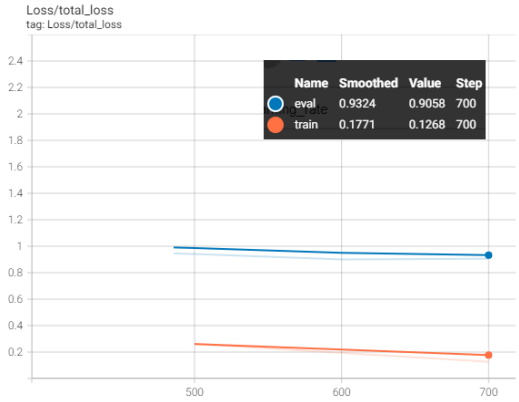

Fig. 4.3 Total loss (training and validation) for smaller version of dataset

Both stated models: Faster RCNN and SSD, being used for object detection purposes, have similarities in terms of design. As described in 2.2 section, detection is still bounded with classification for which we generally use a convolutional neural network. In our case, this CNN will represent the backbone of the network, and this can be changed by finding one which optimally works for a specific dataset. So, in the second step, we compared ResNet50 and RestNet101 as backbones networks for Faster RCNN; MobileNet and RestNet50 as backbones networks for SSD being the ones available in Tensorflow 2. The input size of the images was changed as in the previous case to 512x512 and we interfere also into the batch sizes of the models due to the available RAM restrictions. For Faster RCNN, being a more complex model, the batch size was set to 5 (after this value, GPU ran into an out of memory error) and for the lighter SSD architecture, we used a batch size of 24.

The results after training for 1000 steps are available in *table 4.1*. We considered only the evaluation metrics at step 1000, having a greater impact on our decision. We observe that the models don't have large variations for mean average precision, both in case of precision and recall, but rather in the results of loss.

Model	mAP precision (large)	mAP precision (medium)	mAP precision (small)	mAP recall (large)	mAP recall (medium)	mAP recall (small)	Loss (total)
F-RCNN ResNet50	0.3874	0.1985	0.08602	0.5583	0.3489	0.1619	0.5402
F-RCNN ResNet101	0.3197	0.1508	0.0526	0.4424	0.2722	0.101	0.652
SSD MobileNet	0.4821	0.2796	0.0896	0.6154	0.4469	0.1968	1.315
SSD ResNet50	0.3446	0.2562	0.0992	0.5262	0.4554	0.1838	0.8892

Table 4.1 Performances of Faster RCNN and SDD with different CNN networks

For Faster RCNN, a lighter model such as ResNet50 had a slightly better performance on the chosen dataset and especially the loss had a lower value than in the case of ResNet101 (with almost 10%). SSD had overall better mAP than faster RCNN (due to the larger batch size) but closer when comparing only its backbones networks. Both have a large value for loss but the SSD with ResNet50 will be chosen due to its smaller loss value. Therefore, we will keep ResNet50 for both models that will be moreover investigated.

4.4.2. Model's hyperparameters fine tuning

We have chosen the models for training: Faster RCNN ResNet50 and SSD ResNet50. As these two models have the same convolutional network, tracking their differences during training and comparing the final detector becomes a more accurate choice. Now, a better comparison can be recorded due to their similarities, and we can sense more precise results at the end. For the trainings that will further occur, we will take into consideration a third class (traffic light) and increase the number of steps from the first training. In this way, we will add some difficulties to our training (integrating a new class), but as we want our detector to perform as well as possible during a real time session in traffic, this comes in our advantage. Both networks will keep the default values for hyperparameters as we want to be sure of the overall performance they have. To have a clearer understanding of steps in relation to epochs, a minimum number of steps can be chosen by equation *4.12*, and it will vary from Faster RCNN to SSD networks due to the latter's larger mini batch size. In the beginning of the training, we chose all the images that contained pedestrians and only the cars which were by default in those images, eliminating the images without pedestrians in them.

$$steps = \frac{number\ of\ training\ images}{mini\ batch\ size} \cdot number\ of\ epochs \tag{4.12}$$

This was mainly made due to the unbalanced number of images within classes as cars were utterly exceeding the number of pedestrians or traffic lights (see section *4.1*). So, adding up a new class will increase the number of input images. For 10428 training images and 1 epoch (we want to go through the data at least one time), we will get approximately 2600 steps for Faster RCNN training and 651 steps for SSD. From our observation, we have seen that for the first model 3000 steps will get the loss stabilized, and for the SSD 1200 steps are better from observing changes. Having decided this, we ran one more training session with the above-mentioned steps for each network using the default hyperparameters, so we could have a starting point from which we improve the performances. The *table 4.2* shows the evaluation performances for this starting point which are not the final ones, but the selected checkpoints for a further comparison.

Model	**mAP precision (large)**	**mAP precision (medium)**	**mAP precision (small)**	**mAP recall (large)**	**mAP recall (medium)**	**mAP recall (small)**	**Loss (total)**
F-RCNN ResNet50 ***3000 steps***	0.2545	0.1924	0.1115	0.431	0.3511	0.207	0.6695
SSD ResNet50 ***1200 steps***	0.2785	0.211	0.09172	0.3568	0.3877	0.1497	0.9568

Table 4.2. Checkpoint performances for Faster RCNN and SSD

Before applying changes to the hyperparameters, it is necessary to understand more about our dataset. Working with a large amount of data in general, but especially with images and their corresponding bounding boxes, becomes harder when manipulating it or noticing some of its features. One of the approaches that we used for understanding the characteristics of our objects was to plot a distribution of their aspect ratios and corresponding number of appearances. In the first plot from *fig. 4.4*, we distributed the width over height for each bounding box with their occurrences. For the objects which have a ratio smaller than 1, the height of the object will be larger than the width and, it can be seen, that most of our images have this kind of objects. On the opposite, the ratio that exceeds the value of 1 represent the objects which are wider. Having these graphs done, we can get a glance of the shape of the ground truth boxes and analyze certain aspects based on it. In the second plot, we introduce a scatter chart from which similar properties can be gathered as it shows the distribution of width and height. The fitted line was drawn using regression and, taking the slope value of this line, it can be concluded that data is distributed at the left and

right of 0.5892.

In the following analysis, the distributions will be used for choosing the anchors to help the network know which types of shapes are going to be seen in this dataset and use them as region proposals. In the next sections, the trainings for Faster RCNN and SSD will be divided separately, keeping in mind that some of the chosen hyperparameters will influence differently the behavior for each.

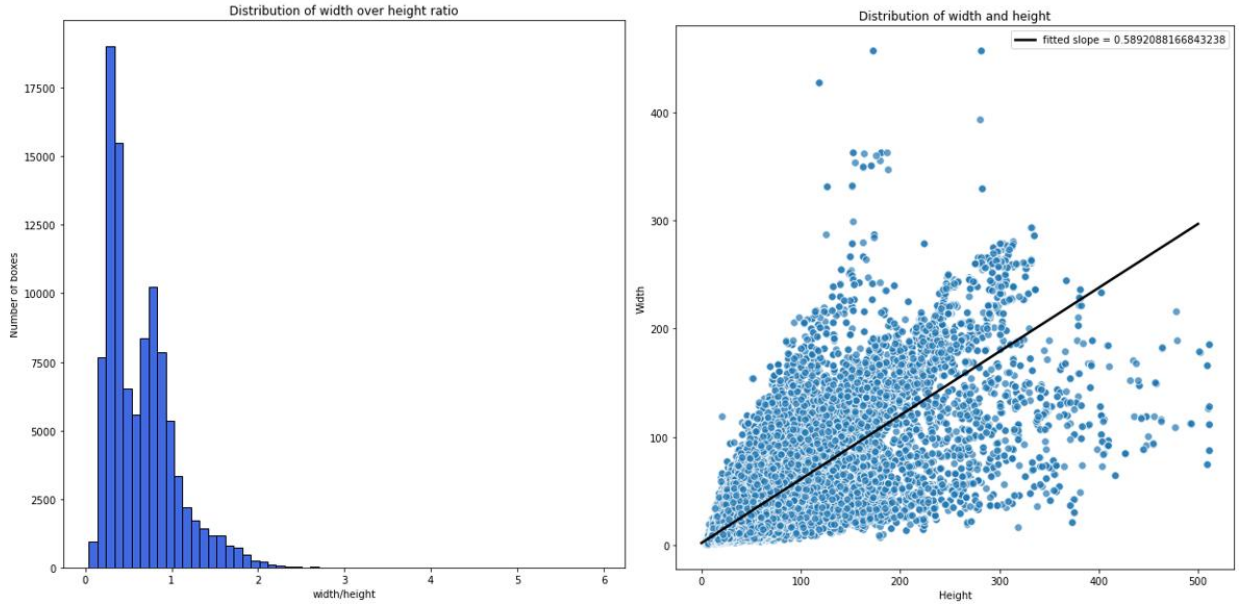

Fig. 4.4. Distribution of width and height

Faster RCNN

Aspect ratio and scales changed, score threshold and IoU increased

The first hyperparameter that was changed for tuning the network is aspect ratio which represents a significant characteristic when helping the model with the structure of the bounding box surrounding the object. As mentioned in the previous section, by evaluating the width-height distributions, we can decide on the most pregnant ratios or at least to be sure that some of them are not even existing in the dataset. Handling more than 100000 bounding boxes without processing them first is not possible in this context then, by evaluating the above distributions, we can see in which range the aspect ratios should be taken. For the first training, we have chosen the following values for aspect ratios based on the previous knowledge gained on our dataset from the distribution and from various trainings: [0.3, 0.4, 0.8, 1, 2]. We chose three values from (0, 1) interval, having most of the objects with these ratios and 1.5 and 2 for covering also the upper bound of the distribution interval. In order to generate the anchor boxes for proposals, we have to consider the scales for aspect ratio vector. The intersection over union, explained in section 4.3.3, was tighten to 0.6 meaning that only the objects that have an IoU over 60% will be considered detections of the same object. This was mainly done to have higher restrictions if we want the detector to be also used in real-time applications. The last adjustment was made to score threshold which was set to 0 in the beginning. This, as mentioned in section 4.3.3, will filter out the

classification proposals so that not every proposal assigned to be accepted. The filter will help the network become more confident when the proposals are classified correctly, so we chose a value of 0.2 for this hyperparameter.

As a first try, we let the default values of scales and trained for 3000 steps with the adjusted ratios. The mean average, for both precision and recall, had an improvement of 20% and 10% respectively for large objects and a small decay (about 4%) for small objects. Considering these results, a smaller value was added to the scales vector for also covering smaller objects in the region proposal stage. This modification led to an even better mAP, each value being increased by 2% from the previous case.

Cosine decay scheduler for learning rate

In terms of learning rate, certain methods can be tried and analyzed, not only trying different values and choosing the best one. Learning rate has the most significant role due to the influence it has on the network's weights. When giving feedback to the network, it will not only be influenced by the prediction along the true output, but also updated with the learning rate. That is why, a large value for this parameter could boost the training, resulting in a small loss in a short time, but not achieving the most optimal solution to our problem. A small value, on the other hand, could slow categorically the process until it stops learning at all. An in-between number for this hyperparameter becomes key in the detector's development and a variation of it could lead to better results. Consequently, even though the optimal learning rate is crucial in these types of applications, letting it constant throughout the training will not train the network at its peak. For encouraging the model's achievements throughout the way, a schedule for increasing its values is almost mandatory to be established. In our case, as a type of scheduler, we chose cosine learning decay. This algorithm quickly increases the parameter and, when reaching a certain threshold, it slowly decays as the execution approaches the end. The strategy behind cosine decay scheduler is to assist the network at the beginning of the training, having more stable weights. When the network reaches a farther step, it means that some of the object's features were retained, and the network can generalize better so there is no need to increase the value of learning rate anymore. For this reason, when the model approaches a certain degree of generalization, given after a specific number of steps, the mentioned value will slowly decrease until it reaches a value very close to 0. This behavior will help when we approach to the global minimum. In the beginning of the training, we want to have larger steps for reaching the minimum but, while we approach, larger steps will not let us converge to the solution wandering around at a much longer distance. What is needed for that stage of the training is to slowly oscillate (with small steps) around the global minimum. In our case, the training improved significantly than in the case of maintaining a constant learning rate, so this scheduler was used for varying it.

Changing optimizer in the first 30,000 steps

For continuing the training, we tackled another hyperparameter which has the greatest significance in how the model will perform and determine certain predictions throughout the way: optimizer. We firstly started by changing the optimizer from momentum to Adam (adaptive moment estimation) optimizer. The former – which was used for the first iterations – helps in eliminating a part of the noise from the loss output signal. It

is a powerful tool which improves stochastic gradient descent (explained in 2.1) and smooths the loss when applied as an optimizer to our model. Momentum will minimize the oscillations which appear in the way of reaching the minimum and will shorten the path, in consequence. What is essential for any type of optimizer is how we choose its parameters, especially the learning rate. The overall aim will be to have a large as possible learning that will not diverge in time and generate a tremendous value for loss. So, choosing the learning rate could decisively influence training in a positive manner only if we find the right tradeoff for this parameter. Nevertheless, various research showed an improved optimizer which presented better results in many applications related to neural network training in image processing field. Adam's success comes mainly from how it was constructed, being a combination of momentum and RMSP (root mean squared propagation) algorithm. The aim of this optimizer, as for any others, is to bring the SGD algorithm to the local minimum as fast as he can. As a combination of two powerful optimizers, Adam should have a quicker response, hence giving better results in a shorter time, having the same purpose as previously described.

For establishing an accurate distinction between momentum and Adam optimizer applied on Faster RCNN model, we firstly took the model trained on momentum optimizer and kept the anchors that gave us the best results beforehand. As for the learning rate, the default values were maintained from the previous iterations with momentum optimizer because they don't generate a huge loss (larger than one), proving to work in this context as the loss hasn't diverged throughout the training. This iteration still helps in choosing the right parameters for the final training, trained only for 30000 steps with each optimizer. So, after 30000 steps (10 times going through the entire dataset), the overall mAP achieved was around 44%, while loss reached 35%. The training loss, after this number of steps, was fluctuating around very different values (from 40% back to 60%) proving that more epochs were needed for the model to learn from the dataset. Consequently, Adam optimizer had a mAP of 39% and an overall loss of also 39%, resulting in poorer metrics than in the previous case. Nevertheless, some observations were derived from this training regarding learning rate and the number of necessary steps for drawing a conclusion.

Adam optimizer, due to its architecture, needed a much smaller starting learning rate. When the model was trained with the same learning rate as for the momentum optimizer (0.01), the loss diverged since the beginning of the steps, being deviated over 2 (more than 100%). As such, more changes needed to be done in this case and the optimal value found for the learning rate decreased to 10^{-3}, while the warmup learning rate became 10^{-4}. The second mentioned value oversees, increasing the learning rate during training after a specific number of steps. However, even with these changes, the step for the previously mentioned cosine decay scheduler had to be increased to 5000 from 2000. The output of these two hyperparameters didn't differ significantly, so the comparison can still take place for more steps. Therefore, we will continue to tune our model for other components and retake this experiment for the complete number steps in order to accurately decide which optimizer works best for our setup and dataset.

Changing second stage localization loss weight

The model was monitored during several trainings throughout the way due to the constant adjustments of the hyperparameters. In consequence, one of the most obvious aspects that was noticed while repeating these steps was the oscillation of loss. In the plots from fig. 4.5, we see that the total loss hardly stabilized during 10 epochs. Even though it fluctuated around same value for a specific number of steps, it rapidly increased in the next iteration. This was determined by the uncertainty of the detector and the small number of batches. Having a closer look at these metrics, we observed that RPN had a stable and small values for both classification and localization (could be seen in fig. 4.5 a and b) and the increasing values were caused by loss from the CNN. As graph c shows, the problem was with localizing the object correctly in the second stage of the detector. This matter strongly influenced the total loss and is the component that we should pay more attention to. Therefore, for the next training we put a higher weight on this second stage localization loss to generate a greater importance for this parameter. The aim of this changing is to draw network's attention to where it should look in terms of detection mistakes.

This localization loss, being in the second stage of detection it corresponds to the Huber loss formula specified in section *4.3.2*. Hence, it is known that for a value which exceeds the normal range for loss (an outlier) mean absolute error will be used for its computation. As such, the error has practically the same value, only assuring its positive

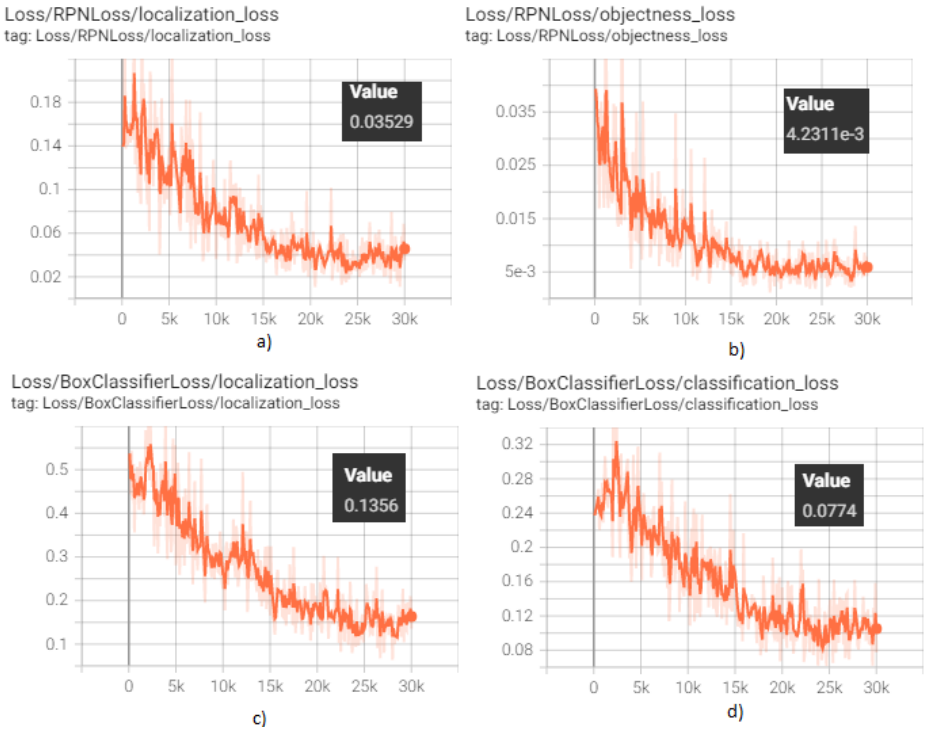

Fig. 4.5 Training loss values after 30,000 steps

nature. What we particularly modify as a hyperparameter for this loss will not influence the formula, but the importance the model should assign for this loss. However, the values which

we add up should be bounded between 0.1 and 0.3 because a larger weight to one loss will influence the other ones and unbalancing problems can be caused. So, after increasing the second stage localization loss weight with 0.1 (for 2 to 2.1), we observed that after 30,000 steps the mAP and loss increased only for large objects and stayed around same values for the other metrics. We then increased this hyperparameter to 2.2, so that more weight to be added during training for the next steps.

Background images

When a detector has more classes which help in identifying the nature of an object, it will also add a supplementary class in charge of identifying the background. So, what is not stated as an object of interest, in our case cars, pedestrians or traffic lights, will be classified as a background object by default. Even though the model is taking this decision by itself when going through the dataset, enhancing this feature could bring better results at the end of the detection. What can be done in this case is to bring images to the dataset which belong to the same environment as it was used to but without any of the objects from another classes. The use of such images becomes useful whenever we want to prevent overfitting and increase the power of generalization of the network by bringing examples how the environment looks like when no object of our interest is present. Due to this powerful technique, we expect to see an improvement specifically for diminishing false positive detections namely, the network should not label objects where they aren't, resulting in a better evaluation output. As a final step of tuning our detector we will add to the training dataset 3500 background images. This number is only a third part from the current dataset, so we expect to see an improvement.

Reintroducing Adam as an optimizer

For the first iterations, our approach in terms of training and evaluating the results was for the sake of comparison of certain hyperparameters which could determine changes from the early stages of the execution. That is why, training for a small number of steps and then comparing the changes worked for the differences which were related to how the model behaved when changing its inputs. As such, selecting other values for anchors' aspect ratios and increasing the threshold for intersection over union gave immediate and visible improvements. For other parameters, such as learning rate, we have seen a rather small increase in performance that is linked to uncertainty of the model at such an early stage. However, having many references and ideas of what parameters work better in general, from various experiments mentioned also in chapter 3, we had an overview on what needed to be changed from the beginning. The experience we gathered using the Adam optimizer at an early stage was understanding that its performances will appear after more epochs, but only with a suitable learning rate. Hence, one of the last adjustments that was made for the final training was changing again into Adam optimizer, but also modifying its parameters for learning rate, as we learnt in the first try. Comparing Adam and momentum optimizers in the beginning for just a few steps, could not lead to conclusive results in terms of optimization differences and a more accurate analysis will be made to the final detector and the untuned one.

Increasing the batch size – noisy loss & changing the parameters for scheduling and learning rate

As suggested in many papers, learning rate still is the hyperparameter which influences the most during training and to the final output. Therefore, we set the learning rate to 1e-3 (an acceptable starting learning rate for Adam) and the warm-up learning rate to 1.33e-4 with increase until it reaches 1e-3 and then slowly end of the training. At first, around 120000 steps were used for training which exceeded 40 epochs and it was thought to be enough for the training. However, two observations were raised regarding this iteration: the model stopped learning after 60000 steps while the loss was stuck to the same value. As it can be seen in fig. 4.6 a, the mAP and loss did not change from step 60000 to step 120000. In plot a, the mean average precision hit a maximum accuracy of 41% and fluctuated around the same value for many steps. In the second plot, we captured both training and validation loss and we can see that we did not encounter overfitting yet, but the model just stopped learning objects' features. More exactly, the loss was converging to a local minimum without increasing in performance. In the second plot, we evaluated the loss until 55000 and again at 120000 because we had seen a pattern for stagnating and the purpose was to verify if this value stays the same until the last step. As a result, two aspects were drawn from this plot, taking into consideration future improvements: the learning rate was too small and the

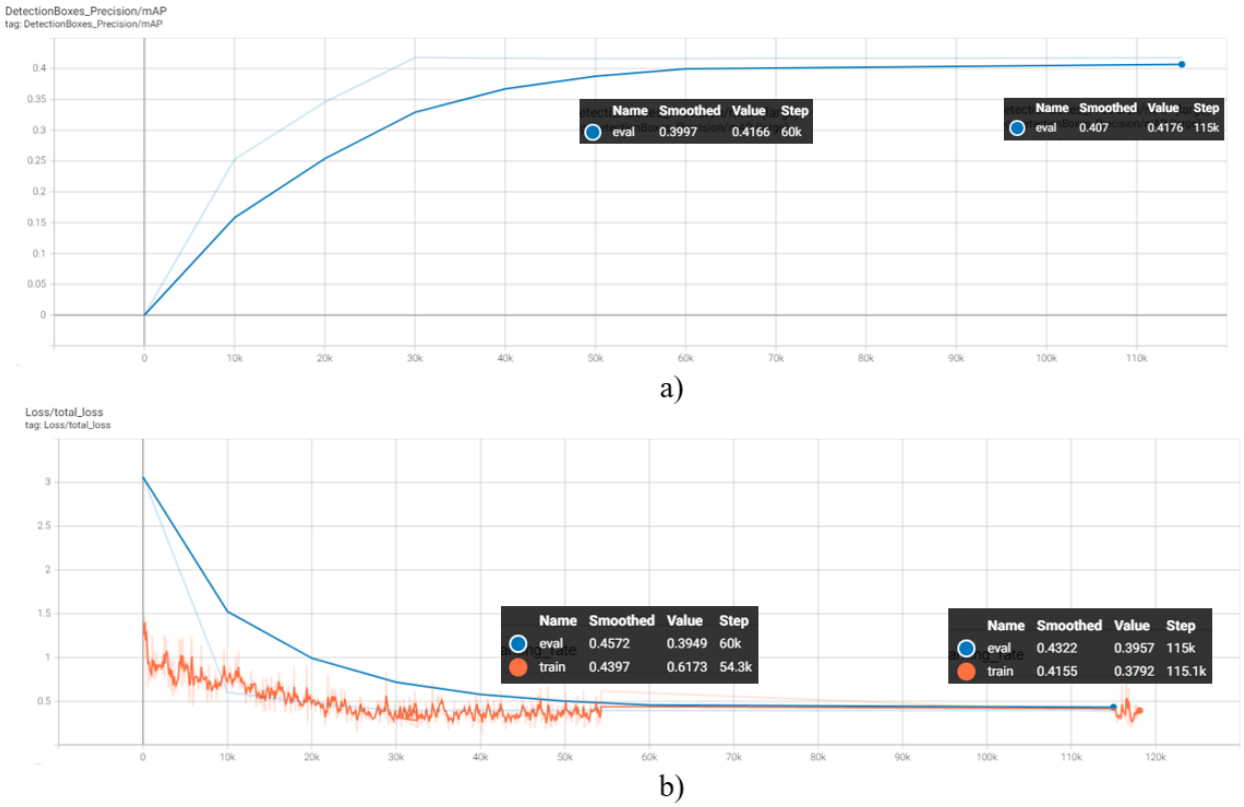

Fig. 4.6 Constant value for mAP and loss

loss was changing its values too rapidly while training. The first supposition was verified in fig. 4.7 where we can see how the learning rate acted during training. As such, it is visible that it reached 0 very soon in our training even before encountering overfitting. As we know,

this scheduler tackles especially the problem of overfitting and the rate should be 0 at the end of the training. A solution for this would be to increase the learning rate while having the same number of decimals, 4e-3 was chosen, and to increase the steps at which the model will reach these values.

The second thing that could be taken from the loss plot are the oscillations that appear from the first steps (which is a normal behavior) but which still acts the same way for all training. Running in parallel the SSD model, which will be covered in the section below, we observed that its loss oscillations were mainly gathering around a certain value. So, the main difference discovered between our detectors was the batch size set in the beginning. Because

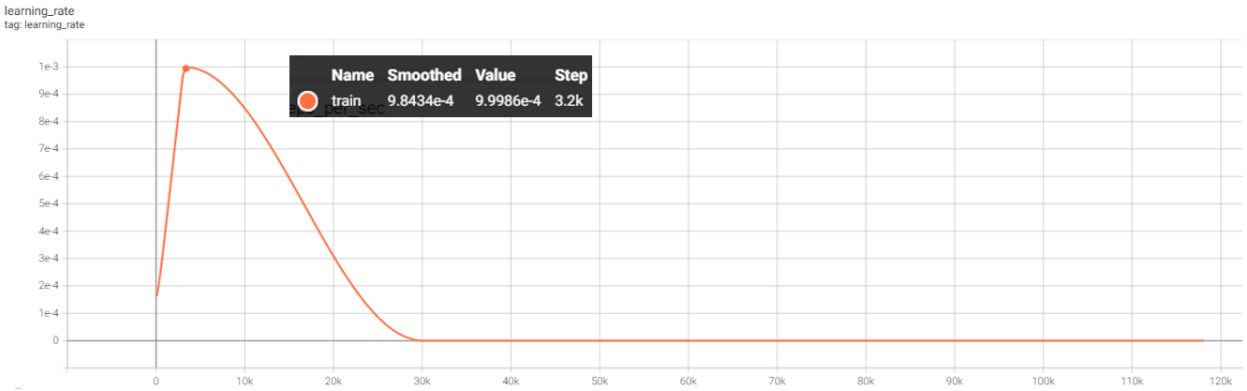

Fig. 4.7 Learning rate decay

of the hardware restrictions, we couldn't set the batch for the Faster RCNN, having a more complex architecture, to more than 5. Therefore, it is difficult for the detector evaluate or give steady feedback while training from small number of examples. For increasing the number of batch size without triggering the hardware, we could use a TPU (also provided by Google Colab) for the next trainings for Faster RCNN. The Tensor Processing Unit is supposed to support larger batch sizes but with smaller execution time. However, increasing the batch size as much as possible (until reaching an out of memory issue) should denote a time of executions similar as in the case of a GPU.

Nonetheless, trying this idea determined a much larger execution time (100 steps per 30 minutes) which becomes unacceptable for our dataset size that needs many epochs to be learned. Therefore, the batch size of 5 was furtherly used in the next trainings. Keeping this batch size, we could only alter the learning rate for better performances. So, we chose the base learning rate for the scheduling to be 1e-3 and the warmup learning rate to 2.5e-4 which, in general, are default rates for Adam optimizer. In fig. 4.8 a, it is emphasized the constant value for validation loss during 30000 steps. Since step 10000, the validation loss remained around 0.7 which is not counted as an improvement of learning. On plot b, the learning rate got to its peak fast, so changing the warmup rate to 1e-4 could solve this problem of loss stagnation.

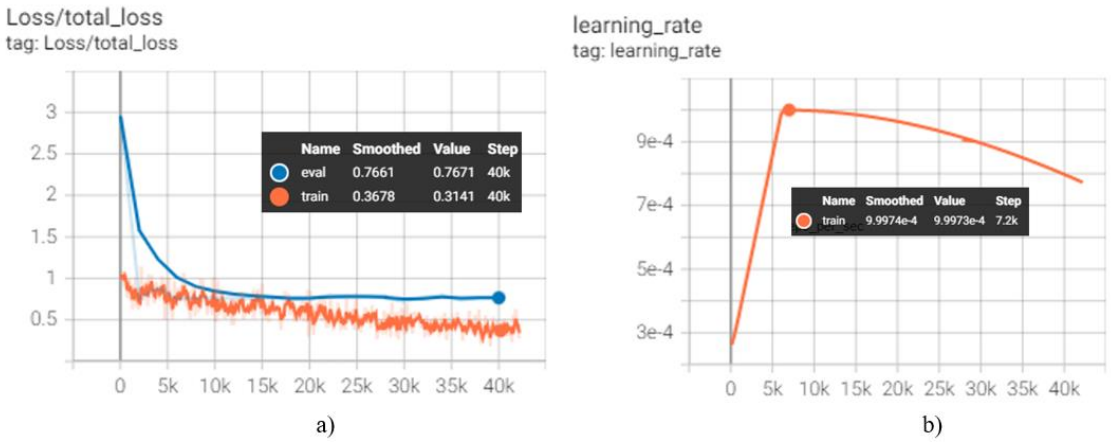

Fig. 4.8 Training and validation loss and learning rate after 40000 steps (Faster RCNN)

For the next training, we set the hyperparameters for learning rate as follows: warmup rate to 1e-5, base rate to 1e-4 and 5000 for warmup steps. These parameters for learning rate seemed to be an in-between approach from what it was seen before in relation to accuracy and loss. Therefore, we decided to train the detector on even more steps (around 140000) from the beginning, keeping the modifications that worked before related to other hyperparameters than learning rate. So, we observed great improvements because the loss got out of the local minimum from before, decreasing by 20%. Also, the accuracy for mAP reached about 57% for COCO evaluation metric for 0.5:0.05:0.95 IoU (which corresponds to an average of performances within this range for intersection over union). The mAP and loss after 148000 steps and also the value for these two metrics at step 114000 (so that we can compare with the achievements from fig. 4.6) can be seen in fig. 4.9. In this scenario, we got rid of the local minimum from the previous step, increasing the accuracy significantly, but we again started to fluctuate around same value without learning anymore. This is due to the same issue of learning loss being brought by the scheduler to 0 too early in the execution. Therefore, we decided to adjust this training by increasing again at step 150000 the value of learning rate to another peak. Also, during this training, the learning rate will also decay little by little, but with a manual scheduler this time. This was a shortest path for increasing the performance without training from the scratch once again. Although, this was not only done due to computational effort, but also because we are not sure if another value for the cosine learning decay scheduler will decrease loss or this is the final lower threshold for it.

In fig. 4.10, we compared side by side the new approach for the learning rate along with the evolution of mean average precision. In plot a, it can be seen the slight increase in learning rate which, eventually, lead to an increasement of 2% of the accuracy. However, after the mAP of 59% was reached, no improvement was seen for another 40000 steps.

Fig. 4.9 mAP and loss for second to last iteration (Faster RCNN)

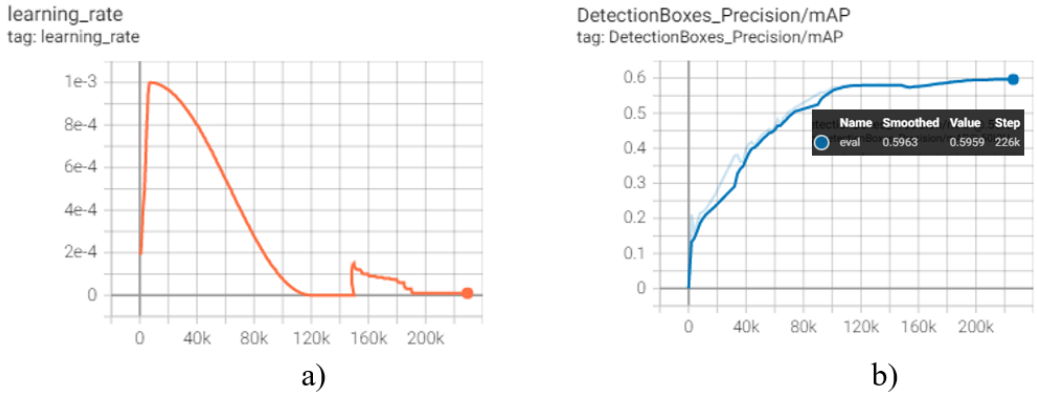

Fig. 4.10 Learning rate and mAP for the final model

From the last step, we understood that the model needs also more steps to stabilize and the scheduling of the learning might need to be adjusted. Even though we reached a high value for performance, we still ran a final iteration, this time, from the beginning because we know the number of steps that are needed. So, having set the warmup rate and base rate as in the previous training, but increasing the number of steps for the scheduling, also increased the performance to 62% mAP. We will consider this to be the best value of the future detector which will be used for testing purposes on the test image form our dataset and on other images that have similarities with our dataset.

Single Shot Detector

Training throughout the entire dataset several times is a time-consuming process and can become disappointing if we don't receive the expected results. Having the previous experience with Faster RCNN, the flow of training and tuning for the Single Shot Detector went smoother because we managed to get rid of some of the mistakes or choices for hyperparameters from the beginning. However, going over a new and distinct architecture, also brought another set of parameters. For the next attempts, we kept similar logic and way of experimenting the performances. In the beginning, we covered the changes on smaller

number of steps and increase them for the hyperparameters which could be observed when the model has learnt basic object features.

Anchors

A one stage detector relies more on how it acts or proposes regions in the beginning of the training. Missing the second stage for detection comes with various advantages (such as faster detection) but with drawbacks as well (poorer detection). The disadvantages can't just be ignored or considered an architecture feature mainly because the mentioned models were created for gaining knowledge faster. So, there are several characteristics which were derived particularly for them. In this case, the right choice for anchors becomes even more significant because it will lead the detector more rapidly to conclusions. The distribution of width per height ratio determined in fig. 4.4 was used for selecting the anchors for SSD. As in the Faster RCNN's training, we decided upon the same aspect ratios for anchors: [0.3, 0.4, 0.8, 1, 2]. As for the scales which are going to be applied over the aspect ratios, the configuration of SSD provides us only one value for manipulating the sizes. Along with the anchors, we adjusted also the hyperparameters from the non-maximum suppression function: intersection over union and score thresholds. Due to the well performances determined on Faster RCNN, we kept 0.6 for IoU and a 0.2 for the filter for classification. The changes were applied to the anchors and, after 3000 steps, we got an increase in performances for most of the metrics, reaching a mAP of 17% and a loss of 59%. Nonetheless, the precision and recall for small objects decreased from what we have seen when we used the default configuration. That is why, we also decreased the scales from 4 to 3 and trained the network from scratch for another 3000. The mAP increased to almost 19%, while the loss reached a lower value, 55%. What was particularly expected was an increase in the metrics related to detection of small objects which, after applying a smaller scale, happened and they metrics got to a larger value than the one received from the default configuration. In the next iterations, we will keep the second training as reference, due to its increased precision, recall and lower number for loss.

Adjusting the focal loss: gamma and alpha

The significance of loss for SSD was covered in 4.3.2. As we know, classification loss took the same approach as in Faster RCNN's using the same formula known as Huber loss. But as the stability of the detector decayed only because it is a one staged model, the localization loss had to be adjusted for having good results in complex detection tasks. Consequently, focal loss was used for localization purposes and, as explained before, had a greater impact on the loss with respect to accuracy. As equation 4.9 shows, a 0 value for gamma will transform this loss into a cross entropy one, but a larger value could reveal the actual advantages of this focal loss. The α parameter from eq. 4.9 determines how much to adjust the output when a class with fewer representations is determined by the probability. Finding the right value for α could help with the problem of an unbalanced dataset. As for our dataset, we do encounter some unbalancing problems, so we decided to increase α to 0.3.

Maintaining the default value, γ=2, and training for 10000 steps with it and the chosen anchors from before, the model achieved a 26% mAP and 50% loss. We increased gamma to 2.2 and, after 10000 steps, got visible better results: 32% mAP and a decrease of loss to 47%.

Along with these metrics, the value of localization loss, particularly, decreased by 3% in the second scenario.

Background images, optimizers and learning rate

For the final iterations, we added 3500 background images, helping the detector decrease its false positive suggestions. For the optimizer, we stick to the Adam due to its advantages for enhancing the model into a faster learning. However, the choice for learning rate had to be tried in many iterations for finding a final value for it. As in the previous case, we kept the cosine learning decay scheduling and modified the parameters for learning rate base, warmup steps, and warmup learning rate. As we encounter the same problem as before, getting a constant loss throughout many steps and a learning rate equal to zero too early (same as in fig. 4.9), we decided to increase the base learning rate to 4e-3. This means that the learning rate will iteratively increase to the base value and then it will take more time to decrease back to zero since it starts decaying from a larger value. However, this strategy determined a growth in evaluation loss reaching over 2 and then a decrease (at step 14000) until 1.4 which is indeed a very high value. Therefore, we changed the base learning rate into 1e-3 but increased the warmup value to 2.5e-4 for reaching the base faster. As it can be seen in fig. 4.11, the losses are very different from each other, training loss fluctuating around 0.5 and training loss being up above 1. As such, for the next training, we have decided to bring hyperparameters for learning rate to a tradeoff, keeping the base learning to 1e-3, the warmup learning rate to 1e-4, but decreasing the warmup steps (the steps after learning rate reaches its peak) to 4000. The decrease of warmup steps is because of the larger number for batch-size (equal to 24) than in Faster RCNN's case which leads to fewer learning steps. As we have chosen to train for 40000 steps, we decrease the warmup steps from learning rate to 4000.

Even though the models were trained in parallel, Faster RCNN, due to its much smaller batch size, had a faster training. That is why, many of the methods that were tried on Faster RCNN, were also tried on SSD, but with the best solutions. During SSD's trainings, several learning rates and even types of scheduling were tried and the results turned out to be small compared to the previous network. In fig. 4.12, we set a cosine scheduling for the learning rate, however, the total number of steps for a full training was thought to be 35000. As the loss was still large and the mAP was around 34%, we decided to use manual

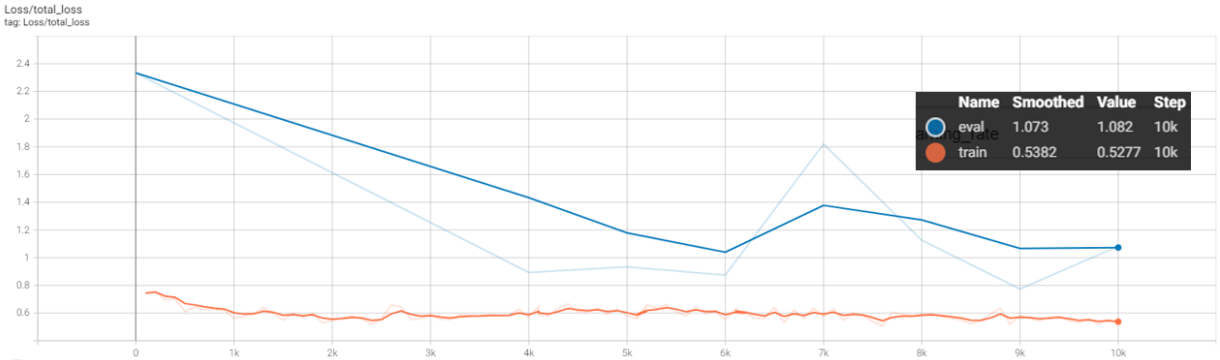

Fig. 4.11 Training and validation loss after 10000 steps (SSD)

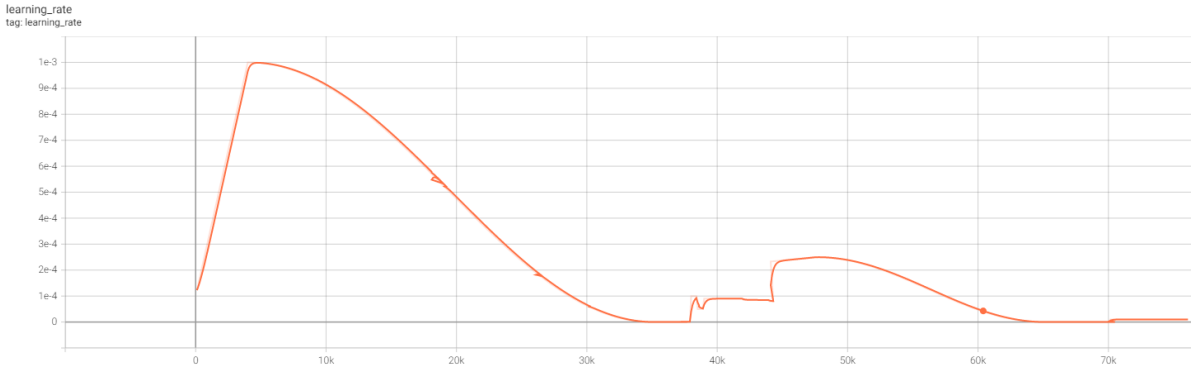

Fig. 4.12 Adjusted learning rate for SSD training

scheduling and to increase little by little the learning rate (from step 35000 to step 45000). As the mAP didn't change much, we decided to retrain with a new cosine scheduling. Sice the training took very long, the strategy was to consider our model trained until 45000 steps a pretrained model and on this one to perform a new cosine learning decay. The mAP did increase to 39% and we will consider it the last iteration for our training.

4.4.3. Comparison of results

Each model we trained for about 80 epochs and was modified during training or started from scratch with a new perspective. As it was clear even while training, Faster RCNN outperformed SSD. That was a result that we expected because of the more complex architecture and due to its region proposal network that takes care of the proposals and their filtering. Lacking this component, SSD can sometimes fail to predict bounding boxes. The results from training the tuned network can be seen in table 4.3. Faster RCNN had much better results, with about 20%. As we decided at the beginning of the training, both networks had the same convolutional network, ResNet50, so that comparison becomes more accurate due to their similarities. Therefore, there is a large gap between these two trained models and that is why we will use Faster RCNN for future real-time detections.

In fig. 4.13, the graphical for mAP is presented. SSD was trained for only 80000 due to its higher batch size, while Faster RCNN was trained for 250000 steps, being trained on only 5 mini-batches. Therefore, for future testing purposes, we will use the Faster RCNN detector for implementing or trying other perspectives in terms of object detection. However, for a future study, we could also analyze why the SSD performances were so low and improve our trainings techniques.

Model	Precision coco	Precision (large)	Precision (medium)	Precision (small)	Recall (large)	Recall (medium)	Recall (small)
FRCNN tuned 250,000	62%	82%	73%	57%	86%	78%	63%
SSD tuned 80,000	39%	65%	57%	32%	70%	64%	39%

Table 4.3 Final results of training

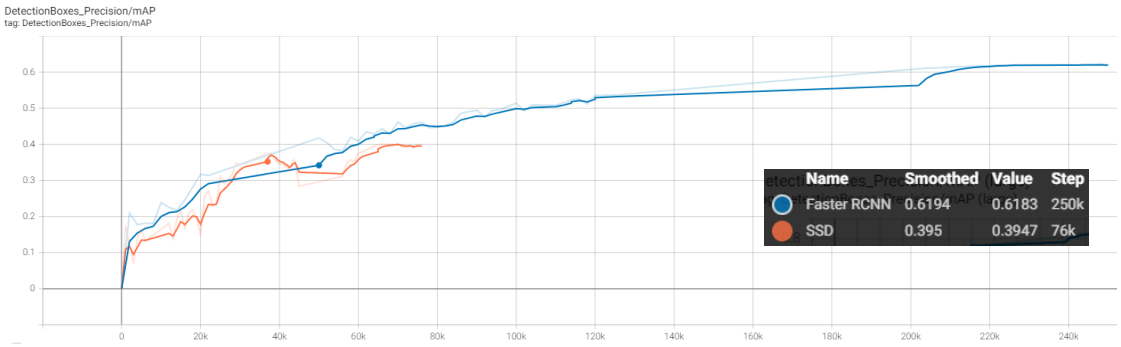

Fig. 4.13 Faster RCNN and SSD performances

Chapter 5. Conclusions

The integration and implementation of deep learning algorithms becomes a complex job because of the many variables that appear in the process. There are many things that have to be thought of from the beginning and, in many cases, experience plays the most significant role. Training a specific dataset, finding the right model and especially finding the right images for your problem are key for a smooth execution. However, training itself comes with its own challenges which have to be solved for continuously increasing the performance.

In this paper, we iterated through many steps regarding the dataset, understanding of the architecture and the intuition behind the algorithms and tuning of the hyperparameters. Beginning the actual training got us through 3 frameworks from which more understanding was gathered related to how are these models set up and what to look for when working with these APIs. The most challenging part was finding a well-balanced dataset with a great number of images and well-done labeling. However, going through the free resources online, you begin to understand the major lack of data and start to find solutions in how to make the most of what you have. Several changes had to be applied on the dataset to make it as balanced as possible. Another crucial point for starting a computer vision project is to be aware of the hardware available, being the component which will influence the most every iteration of this process.

In terms of the dataset, in the stage of preprocessing, we divided it in order to get the most optimal solution in terms of class weighting and managed to rebalance some of the classes that appeared in the images. Even though we partially solved the unbalancing of classes, some labels (observed during validation) were not assigned to some of the objects that did appear in the images. This is, indeed, a serious issue for the future learning because if the network found an object correctly which wasn't labeled, it will wrongly be penalized as the model supposes it detected an object that wasn't there. One of the solutions that could be implemented in the future, would be to use supervised learning for relabeling what was left. This procedure is usually used when an amount of data doesn't have annotations. But in our case, we can use it as a tool to detect what's left for being annotated, taking the already trained model to detect on our dataset. However, it is a more complicated process because we have to be careful which images we choose as inputs from the beginning of the training. If we train it with numerous labels that are not there but they should have been, then we cannot guarantee that that the network learned accurately enough to detect those missing annotations. Supervised learning is based on algorithms that will sense if a detection was confident or not. If it was, we will automatically label the prediction and if not, more investigation has to be done. In order to choose if a prediction on an unlabeled image was successful or not, we need evaluation parameters such as non-maximum suppression.

Tuning a model comes with a great understanding of the network itself and how its mathematical components work. In our representation, based on the several research, we found hyperparameters that were essential to be changed and adapted. For some of them, we tried different values because of the given output, while others worked fine from the early beginning. We didn't try every possible combination because of the long computations,

however for what was more meaningful, such as learning rate, we got through several trainings. One of the popular methods of hyperparameter tuning is given by the greedy method. Using this method, each hyperparameter will be changed with a set of values and the model will be fully trained. This is done for every new changing and it's based only on observation. Even though it may seem an approachable method, the time for computations could take months even if you have a hardware component that performs well. In relation to this, as a general practice, the model is often seen as a black box in which certain parameters are changed and we look only at the performances that resulted from the changes and modify again the input. That can be true for some of the inputs because it might influence much differently than it was supposed to. But understanding the concepts and gaining the intuition behind its architecture can ease the process and help in identifying several problems in the training.

A future approach in terms of deployment of the detector would consist in using it for various real-time applications. As such, the detector can be used in the car and adapt the distance between it the car in front as in the cruise-control system. However, for this kind of application, bounding boxes are not the best solution of identification. If we had estimated the distance by the size of the bounding box, we would have estimated badly if a larger car is in front of us. A solution or a better practice would be to use a Mask RCNN model because it will use the edges from the car and better judgement regarding the distance can be determined. As presented before, many paths can be taken for increasing the accuracy of the detector. Additionally, it can be used for many reasons in real-time and improved based on how it acts while we use it.

Bibliography

[1] G. Ciaburro and B. Venkateswaran, *Neural Networks with R: Smart models using CNN, RNN, deep learning, and artificial intelligence principles*. Packt Publishing Ltd, 2017.

[2] "What is a Neural Network?," *TIBCO Software*. https://www.tibco.com/reference-center/what-is-a-neural-network.

[3] N. Kriegeskorte and T. Golan, "Neural network models and deep learning," *Curr. Biol.*, vol. 29, pp. R225–R240, Apr. 2019, doi: 10.1016/j.cub.2019.02.034.

[4] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights Imaging*, vol. 9, no. 4, pp. 611–629, Aug. 2018, doi: 10.1007/s13244-018-0639-9.

[5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks." arXiv, Jan. 06, 2016.

[6] W. Liu *et al.*, "SSD: Single Shot MultiBox Detector," vol. 9905, 2016, pp. 21–37. doi: 10.1007/978-3-319-46448-0_2.

[7] I. S. Mohamed, "Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques," 2017. doi: 10.13140/RG.2.2.30795.69926.

[8] C. C. Aggarwal, *Neural networks and deep learning: a textbook*. Cham, Switzerland: Springer, 2018. doi: 10.1007/978-3-319-94463-0.

[9] "Papers with Code - Max Pooling Explained." https://paperswithcode.com/method/max-pooling

[10] K. R. Akshatha, A. K. Karunakar, S. B. Shenoy, A. K. Pai, N. H. Nagaraj, and S. S. Rohatgi, "Human Detection in Aerial Thermal Images Using Faster R-CNN and SSD Algorithms," *Electronics*, vol. 11, no. 1151, p. 1151, Apr. 2022, doi: 10.3390/electronics11071151.

[11] J. WANG, X. Niu, P. ZHANG, Y. Dou, and F. Xia, "Aircraft Detection in Remote Sensing Images via CNN Multi-scale Feature Representation," *DEStech Trans. Comput. Sci. Eng.*, Jul. 2017, doi: 10.12783/dtcse/smce2017/12424.

[12] D. R. Chowdhury, P. Garg, and V. N. More, "Pedestrian Intention Detection Using Faster RCNN and SSD," in *Advances in Computing and Data Sciences*, Singapore, 2019, pp. 431–439. doi: 10.1007/978-981-13-9942-8_41.

[13] P. Garg, D. R. Chowdhury and V. N. More, "Traffic Sign Recognition and Classification Using YOLOv2, Faster RCNN and SSD," 2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2019, pp. 1-5, doi: 10.1109/ICCCNT45670.2019.8944491.

[14] J. -a. Kim, J. -Y. Sung and S. -h. Park, "Comparison of Faster-RCNN, YOLO, and SSD for Real-Time Vehicle Type Recognition," *2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*, 2020, pp. 1-4, doi: 10.1109/ICCE-Asia49877.2020.9277040.

[15] Y. al Atrash, M. Saad, and I. H. Alshami, "Detecting and Counting People's Faces in Images Using Convolutional Neural Networks," in *2021 Palestinian International Conference on Information and Communication Technology (PICICT)*, Sep. 2021, pp. 116–122. doi: 10.1109/PICICT53635.2021.00031.

[16] Q. Zhang, X. Chang, and S. B. Bian, "Vehicle-Damage-Detection Segmentation Algorithm Based on Improved Mask RCNN," *IEEE Access*, vol. 8, pp. 6997–7004, 2020, doi: 10.1109/ACCESS.2020.2964055.

[17] "COCO - Common Objects in Context." https://cocodataset.org/#detection-eval.

[18] S. J. (Stuart J. Russell, *Artificial intelligence: a modern approach*, Fourth edition. Pearson, 2022.

[19] R. Atienza, *Advanced Deep Learning with TensorFlow 2 and Keras: Apply DL, GANs, VAEs, Deep RL, Unsupervised Learning, Object Detection and Segmentation, and More, 2nd Edition*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited, 2020.

[20] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal Loss for Dense Object Detection," 2017, pp. 2980–2988.

[21] J. Hosang, R. Benenson, and B. Schiele, "Learning Non-maximum Suppression," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, Jul. 2017, pp. 6469–6477. doi: 10.1109/CVPR.2017.685.