TU WIEN Informatics

Nighttime Object Tracking for Intelligent Vehicles

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Elias Frantar, B.Sc.
Matrikelnummer 01527171

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz

Wien, 15. Oktober 2020

Elias Frantar	Margrit Gelautz

TU WIEN Informatics

Nighttime Object Tracking for Intelligent Vehicles

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Elias Frantar, B.Sc.
Registration Number 01527171

to the Faculty of Informatics

at the TU Wien

Advisor: Ao. Univ. Prof. Dipl.-Ing. Mag. Dr. Margrit Gelautz

Vienna, 15th October, 2020

Elias Frantar	Margrit Gelautz

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Elias Frantar, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Oktober 2020

Elias Frantar

Acknowledgements

First and foremost, I want to thank the Austrian Research Promotion Agency (FFG) and the Austrian Federal Ministry BMVIT for funding this thesis as part of the CarVisionLight project under the program "ICT of the Future" (project no. 861251). Then I want to thank ZKW for giving me the opportunity to work on an exciting and at the same time also highly relevant real-world problem, in particular, also for providing various interesting nighttime driving datasets with several unique properties. Further, I want to thank our second project partner, emotion3D, for insights into their related work and useful suggestions during project meetings.

For the people at TU Wien, I want to thank Johannes Fromm, who swiftly resolved all kinds of IT related issues and ensured that working from home during the pandemic went smoothly. Next, I want to thank Florian Groh, who was responsible for annotating and curating the CarVisionLight dataset, which was central to the development and evaluation of most methods described in this thesis. Further, he took on the very tedious task of readjusting the dataset labels several times to make sure that the performance measurements in my work were as representative of real-world performance as possible. I also want to thank Dominik Schörkhuber for his good ideas during project discussions as well as for exploring orthogonal research directions, which helped me in deciding what to try next.

Last, but certainly not least, I want to thank my supervisor Professor Margrit Gelautz, for excellently organizing and coordinating the project, for the many fruitful discussions, and for the highly detailed feedback that greatly elevated the quality of this thesis.

Kurzfassung

Selbstfahrende Fahrzeuge haben großes Potential und sind deshalb in letzter Zeit eine besonders aktive Forschungsrichtigung. Trotz großer Fortschritte, im Speziellen durch neue mächtige Techniken des maschinellen Sehens basierend auf konvolutionalen neuronalen Netzwerken (CNNs), ist eine weite Verbreitung von vollständig autonomen Fahrzeugen wahrscheinlich noch mindestens ein paar Jahre entfernt. Eine deutlich unmittelbarere Anwendung von verwandten Methoden sind fortgeschrittene Fahrassistenzsysteme, welche mit dem Menschen am Steuer zusammenarbeiten, z.B. durch Warnungen vor schlecht ersichtlichen Hindernissen. Besonders bei anspruchsvollen Fahrbedingungen, wie in der Nacht oder bei schlechtem Wetter, könnten solche Unterstützungstechnologien einen wertvollen Beitrag zu erhöhter Fahrsicherheit leisten.

Im Rahmen dieser Diplomarbeit werden Techniken erforscht, implementiert und evaluiert, welche die Basis für ein Nachtsichtsystem für intelligente Autos bilden könnten. Genauer gesagt, betrachten wir das Problem des zwei-dimensonalen multiplen Objektrackings in Echtzeit angewandt auf Videos, welche von Fahrzeugkameras bei Nachtfahrten im Rahmen des CarVisionLight-Projekts (CVL) in ländlichen Umgebungen aufgenommen wurden. Wir ermitteln, wie gut aktuelle Tageszeitobjektdetektoren nachts funktionieren. Wir untersuchen, wie große bestehende Verkehrsdatensätze optimal dazu genutzt werden können, einen CNN-basierten Objektdetektor zu trainieren, welcher besonders effektiv für die deutlich anderen CVL-Daten ist. Dann erweitern wir dieses Detektionsmodell zu einer vollständigen Trackingmethode, wobei wir einige speziell auf unseren Anwendungskfall abgestimmte Erweiterungen integrieren. Schließlich kombinieren wir all unsere Erkenntnisse, um einen Prototypen für ein selbständiges Trackingtool zu entwicklen.

Zu den interessantesten Entdeckungen im Zuge unserer Untersuchungen gehören, dass (1) etablierte allgemeine Tageslichtmodelle ungeeignet für CVL-Daten sind, (2) Training mit nur 25% der Berkeley Deep Drive Bilder bereits 0.48 Validierungs-mAP liefert (verglichen mit 0.5 für alle Daten), (3) primär auf Nachtdaten trainierte Modelle auch gut untertags funktionieren und (4) ein Finetuning auf Bildern mit höherer Auflösung zu 0.07 mAP Inferenzverbesserungen führt. Weiters führen die domänenspezifischen Trackeradaptierungen zu einem signifikaten Anstieg der Trackingkonsistenz (1/3 weniger ID-Switches und 14% höherer Recall). Schlussendlich zeigt eine quantitative sowie qualitative Evaluierung, dass unser Tool eine effektive Lösung für die behandelte Problemstellung darstellt, sowohl bezüglich Tracking-/Detektionsqualität als auch Ausführungszeit, und Potential für zukünftige Weiterentwicklung bietet.

Abstract

Self-driving vehicles hold great promise and therefore have become a very active area of research in recent years. Although tremendous progress has been made, in particular through new powerful computer vision techniques involving convolutional neural networks (CNNs), widespread adoption of fully autonomous vehicles is likely still at least a few years away. A more immediate application of related methods are advanced assistance systems that work in conjunction with human drivers, for example, by warning them about difficult-to-spot obstacles. Especially under challenging driving conditions, like during the night or in bad weather, such support technologies could make a valuable contribution towards safer traffic.

This thesis explores, implements and evaluates techniques that could form the basis of a night vision system for an intelligent car. More concretely, we study the problem of real-time two-dimensional multiple object tracking in videos, recorded by an in-vehicle camera, as a part of the CarVisionLight (CVL) project, while driving in rural areas at night. We assess how well state-of-the-art daytime models generalize to a nighttime setting. We study how to optimally utilize big existing driving datasets for training a CNN based object detection model that is particularly effective on the considerably differing CVL data. Then, we extend this detection model to a full tracking method while also applying several enhancements aimed at improving the performance in our specific setting. Eventually, we put all of our insights together and develop a prototype for an end-to-end tracking tool.

Among the most significant findings of our various investigations are that (1) established general purpose daytime models are not suited for CVL data in our experiments, (2) training with only 25% of the images in the Berkeley Deep Drive dataset yields already 0.48 validation mAP (as opposed to 0.50 when training with the full dataset), (3) a model trained primarily on night data can also perform well during the day and (4) finetuning on higher resolution images leads to a mAP improvement of 0.07 at inference time. Further, our domain specific tracker adaptions provide a noticeable increase in tracking consistency (3 times less ID-switches) and recall (14% higher). Lastly, an extensive quantitative and qualitative evaluation shows that our tracking tool poses an effective solution to the problem at hand, both in terms of tracking/detection quality as well as in terms of execution speed, and has potential to stimulate future work.

Contents

CHAPTER 1

Introduction

Cars that are able to drive safely without the assistance of any human driver have been a dream of many for decades. Dramatically reducing the number of accidents by eliminating human error, saving large amounts of time by allowing people to focus on other tasks while commuting to work and helping the environment by making car-sharing a lot more appealing, they pose enormous promise for society. Naturally, much research related to such "self-driving vehicles" is currently happening all around the world, not just in academia but also directly in the industry. With recent advancements in computer vision, especially through convolutional neural networks, and powerful embedded hardware, huge progress towards the ultimate goal of fully autonomous cars has been made in the past few years.

While opinions on when the first actual self-driving vehicles will be ready (and even what really constitutes "full" autonomy) differ widely, it is clear that at least mass market adoption will still take quite some time. However, many of the sub-systems making up an autonomous car are not just useful when it is driving completely on its own, but also to support a human driver. Further, such assistance systems do not require perfect accuracy as the car is still being controlled by a person that can take over control in critical situations. State-of-the-art vision systems are not yet perfect and sometimes still make obvious errors, but they are in principle ready to provide very useful help to drivers. In fact, several assistance systems enabled only by cutting-edge vision technologies have already found their way into commercial cars: automatic lane keeping, parking or high-beam light control as well as detecting obstacles when driving backwards, to name but a few. These and other new, even more intelligent assistants are only going to be become more and more popular in future and thereby contribute significantly to safer traffic long before self-driving vehicles will be available for widespread use.

One particularly important area for advanced assistance systems is driving in the dark. Not only is safely controlling a car during the night hard and requires great attention by a human driver, but it is also a situation many people are confronted with on a daily basis

when commuting to/from work early in the morning or late in the evening. A robust nighttime vision system could warn the driver about difficult-to-spot obstacles or directly connect to the vehicle's lighting system and intelligently illuminate the environment based on detected objects. This could include automatically adjusting the high-beam lamps to always ensure maximum visibility without blinding other drivers. Further options would be to automatically shine on traffic signs to make them easier to read or even preemptively highlight difficult-to-spot obstacles such as a pedestrian that is anticipated to cross the road. Certainly, such a system would make driving at night a lot more comfortable and reduce the number of accidents.

Since vision in a highly dynamic environment with a large number of actors, like heavy city traffic, is already extremely challenging, the majority of related publications refrains from introducing even more difficulty via bad weather or unfavorable lighting conditions. Unfortunately, even systems that perform very reliably with good sight typically degrade rather quickly once conditions are less than optimal. Only very recently more research is starting to specifically consider scenarios such as nighttime, rain, fog or snow.

The importance of the problem coupled with the fact that it is not yet well covered in literature is a major motivation for this diploma thesis, in which we want to explore the development of computer vision components for a nighttime object tracking system to be used by intelligent vehicles.

1.1 Problem Statement

Building a reliable general purpose night vision system for cars is a challenge which will most certainly require the joint effort of many big research teams to solve fully. Hence, we can only address a small sub-problem within the scope of this thesis. This will be real-time 2D object tracking in videos recorded by a standard RGB-camera located in a driving car with a focus on detecting other vehicles.

Using a standard camera for nighttime vision may seem a bit counter-intuitive when much less light reliant technologies like LIDAR or night-sight devices exist. Those sensors definitively hold great promise but also have major shortcomings, such as their limited range, unreliability in bad weather and high prices. Further, current computer vision algorithms usually have been developed for normal camera recordings and the majority of publicly available datasets also consists of standard images. Since experienced human drivers can drive well at night based exclusively on what they see, an advanced enough computer vision model should, at least in theory, be able to do the same (i.e. we are not dealing with an inherently ill-formed task here). All in all, we can safely conclude that camera-based night vision is a very relevant problem that reaches clearly beyond the current state-of-the-art.

Probably the first thing that comes to one's mind when thinking about a nighttime assistance system is detecting "obstacles" like other vehicles or pedestrians. However, merely detecting them is not sufficient for many applications; rather we also need to

understand how they move over time. This is especially true when detection models are not yet 100% reliable and still occasionally miss objects at seemingly random times; just imagine the computer thinking that a car driving towards you has completely disappeared whenever the detector drops a single frame. Hence, our work will consider not only detecting objects but also consistently tracking them over time. At first instance, the most important "objects" to consider are other vehicles, driving cars for obvious reasons, but also parking ones as passengers may be just about to exit their vehicle and since careless pedestrians can appear between them. In fact, the latter situation is particularly interesting as it cannot be addressed at all by classical vision methods that extract just the lights of other cars (see Section 2.1). An ideal system would of course also accurately detect pedestrians, traffic signs and even completely unexpected obstacles blocking the road. To maintain a realistic goal for this thesis, we will focus primarily on detecting and tracking vehicles, additional object types and other extensions will still be considered but not explicitly optimized for.

A perfect system would not just locate objects in a video stream but fully infer their real-word shape and 3D position. However, despite enormous progress in vision techniques, this is still extremely difficult to do with high accuracy, even more so in unfavorable conditions such as in the dark. Hence, this work will concentrate on detecting objects by their 2D bounding boxes. Doing so already requires the model to have a solid understanding of object shapes and sizes and is thereby an important intermediate step towards full 3D-tracking. Note also that reliable 2D information would already be extremely useful, especially coupled with coarse distance estimates calculated from camera parameters.

While there is thus far generally only little prior work dealing specifically with nighttime object tracking, our work is particularly motivated by data recorded as part of the CarVisionLight project. These videos almost exclusively depict scenes in rural areas or villages. In contrast, the majority of traffic related research, in particular also the big open datasets that most publications use for evaluation, focuses on city or highway scenarios. Especially at night, these differ significantly from our scenes. It is darker in rural areas as there is less ambient light, the driving speed is usually higher than in the city and detecting small far-away objects is critical. The latter needs to be highlighted because it is a) very challenging for powerful deep learning based detection models, even more so in real-time, and b) not so important in city/highway scenarios as there tend to be more immediate threats that are a lot closer to the camera. A more detailed discussion of the dataset discrepancies can be found in Section 2.4. We will study how to optimally leverage the existing datasets for learning models that perform well in our noticeably different domain.

To summarize, this work is about nighttime camera-based 2D object tracking optimized for driving in rural areas, a very relevant but thus far little examined problem (especially in the context of recent advances in deep learning methods).

1.2 Aim of the Work

This thesis has several goals. Concretely, we hope to answer the following list of questions:

1. How do state-of-the-art deep learning based object detection models trained on daytime datasets perform in nighttime traffic situations?
2. What is an effective way to utilize big existing driving datasets with significant amounts of night images to improve performance at night, especially in rural areas which are not well covered by the training data?
3. How can one extend such a nighttime detection model to also reliably track objects across time, in particular, how can we adapt existing methods to compensate for common issues we observe in our concrete application?
4. Can we build an accurate but still real-time capable tracking program based on the insights we gained from studying the previous questions?

To do so, we will conduct extensive experiments and perform quantitative but also qualitative evaluations as well as detailed comparisons and interpretations. Eventually, the goal is to put together all our findings to implement a simple, yet complete in terms of core features, end-to-end tracking program dedicated for our use-case. We hope that our results will be useful in the context of the CarVisionLight project and perhaps even bring us a little closer to the next generation of nighttime driver assistance systems.

1.3 Structure of the Thesis

The thesis begins with a detailed overview of the state-of-the-art techniques in object detection and tracking with a focus on recent advanced but still already proven ideas. Early approaches to nighttime vehicle tracking as well a discussion on datasets and other related research areas are included as well. Chapter 3 continues by describing key implementation aspects, the end-to-end tracking tool and the evaluation in-depth. Chapter 4 presents, analyzes and compares the results of all our experiments before documenting the final evaluation of the developed tracking software. This work ends by summarizing the most important results, drawing conclusions and outlining various potential future research directions.

CHAPTER 2

State of the Art

The aim of this chapter is to provide an overview of major techniques relevant to the work of this thesis. First, we look at some early nighttime tracking systems using classical computer vision techniques. Next follows and in-depth discussion of the current state of the art in deep learning based object detection and tracking. Afterwards, we describe several relevant driving datasets containing significant amounts of night data. The chapter concludes by outlining a few other interesting related research areas.

2.1 Early Approaches to Nighttime Vehicle Tracking

Computer vision had already been a very active area of research long before the nowadays omnipresent deep neural networks became popular in recent years. Similarly, early camera-based systems for tracking vehicles during nighttime already appeared many years ago, for example, in the form of the solutions by Alcantarilla et al. [ABJ+08] (and their follow-up work [ABJ+11]) or Salvi [Sal14]. We will now go on to discuss those approaches in more detail. Figure 2.1 provides an overview of Alcantarilla et al.'s system, which is discussed in more detail throughout the next several paragraphs.

Since humans need light, of which there is only very little during the night, to see, cars (especially newer ones) feature several strong light sources. These not only illuminate the immediate environment but also present strong cues that allow the driver to recognize other vehicles from far away. Analogously, computer vision systems can also try to detect cars by their lamps, which is exactly what the approaches we discuss in this section are doing. At first glance, this might even seem easier than automatically detecting cars during the day, where no such simple but strong indicators are available. However, a closer look reveals that reflections, traffic signs and distracting light sources make it not straight-forward to extract just the car lights.

Alcantarilla et al. [ABJ+08, ABJ+11] apply adaptive thresholding (adaptive rather than global to account for inhomogeneous lighting conditions throughout the image) to extract bright blobs in a gray-scale video. In similar fashion, Salvi [Sal14] initially transforms the RGB-image to the LAB color space (this separates the brightness of a color L from its hues A and B) before also doing adaptive thresholding on the L-component followed by morphological opening and closing to remove noise and smooth out boundaries. In both cases this process results in a set of bright image regions. The big challenge now is to figure out which of them are really car lights; in this point the two approaches differ principally, as explained in the following.

First, blobs in consecutive frames with small Euclidean distance are matched together and then considered as the same object (or track) by Alcantarilla's system. Additionally, a coarse distance estimate is obtained through a perspective camera model. Finally, objects are classified with a support vector machine (SVM) model operating on an experimentally determined set of features such as pixel area, contour length, circularity or the vertical centroid coordinate. The authors report good performance for detecting head-lights even at a few hundred meters away, but only mediocre performance for tail-lights (just 50-80 meters), which they attribute to the generally lower luminance and much higher variability in terms of shape of the latter lamps. Salvi's system, which was originally designed for the surveillance of a highway by a stationary camera, works differently. Instead of applying a learned classifier, non-headlights are filtered out via heuristic rules. Blobs outside of predefined images regions (with the general layout of the scene known in this case) as well as those that do not meet the circularity constraint are discarded. That constraint is judged by the aspect ratio of the blob's bounding box being too far away from 1 and the fraction of bright pixels within the box being too small. The next step is pairing head-lights based on their size and horizontal alignment to identify individual vehicles. Tracking happens by matching detected vehicles in consecutive frames that are both close in Euclidean distance of their centroids as well as in size (box area). An evaluation of the whole system carried out by the authors for the task of counting cars yields well above 95% detection rates with only few false positive. However, it needs to be noted that counting the number of cars which pass over a highway just requires detecting each car for a few frames once it is close enough, which can be accomplished more easily than accurately tracking all vehicles in the video.

2.1.1 Shortcomings

While both approaches discussed in the previous section already work quite well for the use-cases they were designed for, extending them to more general night vision systems may not be feasible. In fact, many of their major shortcomings in the context of our application are inherent to the strategy of relying exclusively on vehicle lights. We will now discuss these in more detail.

To start off, car lights can only be reliably extracted via thresholding if they are indeed always among the brightest spots in the image. This is highly dependent on the camera in use as well as the regarded scene. This is also the reason why Alcantarilla et al. use highly

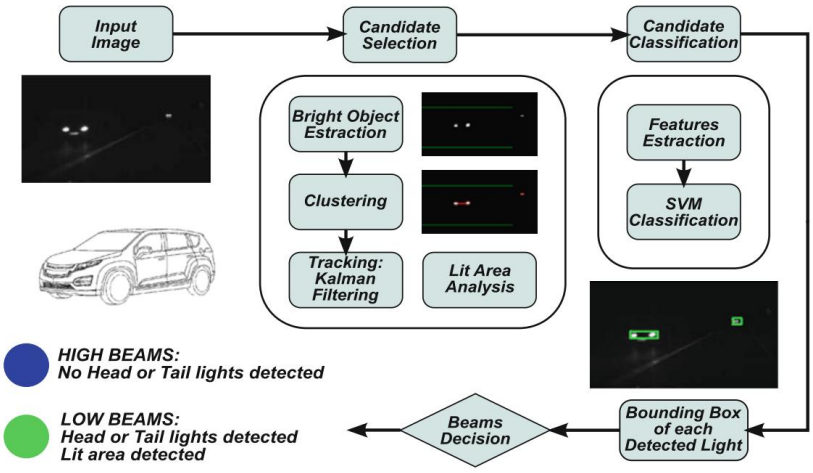


Figure 2.1: Overview of Alcantarilla et al.'s light-based vehicle-tracking system. Figure from [ABJ+11]

light-sensitive black and white cameras. Still, they report problems in reliably detecting the far less luminescent rear-lights of preceding vehicles. Further, they only apply their algorithm in very dark scenes, basically without any non-car light-sources. While this is very much a sensible approach for their use-case, i.e. an automatic high-beam controller, this would be a major problem for a more general night vision system. Another issue is that pure light-based detection can only really detect the presence and approximate location of a car rather than inferring full bounding box information from just the noisy white blobs, which would be extremely difficult. For far-away cars this may not be an issue; however, once cars get closer more precise location information would certainly be useful. Perhaps the most profound limitation of detecting objects solely by their lamps is that obviously things that do not have their own light-sources are excluded from detection. Parking vehicles, a deer jumping in front of the car, pedestrians walking close to the border of the road or a fallen tree blocking the way will all be completely invisible to the vision system. Additionally, while it might be theoretically feasible to detect trucks, motorbikes, signs and traffic lights, reliably distinguishing between them would be tricky if the only available cues are a few bright spots in the image.

All in all, it can be said that detecting driving vehicles by extracting their lamps is an idea which has its merits. The literature shows that it works surprisingly well for a few narrow (although certainly important) applications [LHB+08, Che09, OGJ08], but it is clearly insufficient to act as the basis for more general purpose night vision. Safely navigating autonomous vehicles during the night as well as advanced driver assistance systems require more comprehensive techniques, which will be the focus of the remainder of this chapter.

2.2 Modern Object Detection Techniques

Started by AlexNet [KSH12] winning the ImageNet Large Scale Visual Recognition Challenge back in 2012, deep convolutional neural networks, henceforth called CNNs for short, have been taking over the field of computer vision by storm. The idea of addressing the immense complexity required to visually understand the real world by "simply" having the computer figure out its own way to do it based on large amounts of data as reference solutions has proven so powerful that it nowadays outperforms manually engineered solutions on almost all key vision problems. In particular, this is also the case for object detection, the task of locating (typically by drawing their bounding box) objects (e.g. people, animals, cars, etc.) while often also identifying their class (e.g. whether an animal is a cat or a dog). Figure 2.2 shows a typical output of a modern object detector. A dog, a bicycle and a car were correctly detected and marked with their bounding box.

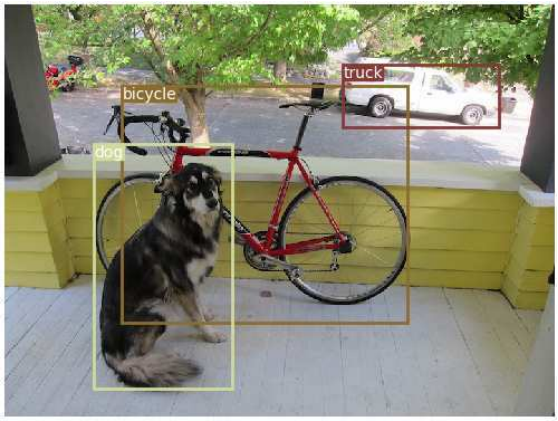


Figure 2.2: Example detection of a modern object detector, `https://github.com/eriklindernoren/PyTorch-YOLOv3` (accessed 2020-07-25).

We start off the remainder of this section with a brief general introduction to deep learning methods before focusing on their application to the task of object detection in particular. There we begin with a discussion about how to properly assess object detection performance. Then we explain various key ideas, techniques and tricks powering current systems. Finally we conclude with a short overview about the state-of-the-art results on particularly relevant standard benchmarks.

2.2.1 Deep (Convolutional) Neural Networks

This section presents a short introduction to the fundamentals of deep learning / deep neural networks. The goal is to provide the background necessary for following the discussions throughout the remainder of this chapter. An in-depth treatment of those topics can be found, for example, in [GBC16].

On a high-level, a *neural network* is a composition of functions f_i for $i = 1, \dots n$ as shown in (2.1). The input of f_1 is $x \in X$ and the output of f_n is $y \in Y$, hence the whole network $F : X \to Y$ is a mapping from X to Y. In this context, individual functions f_i are typically referred to as *network layers*. A *deep neural network* is simply a neural network with a "large" (there is no formal quantification) number of layers n.

$$y = f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = f_n(f_{n-1}(\cdots f_1(x))) = F(x) \tag{2.1}$$

Further, each function f_i is parametrized by its corresponding *network weights* called w_i. We denote the set of all w_i as W. Now, the key idea is that these weights are not fixed in advance but rather determined "automatically" by *learning* from data. More concretely, given a dataset D consisting of (x_j, y_j) pairs, we want to find a set of weights such that the neural network produces the corresponding y_j when given an x_j. This is accomplished via a *loss function* $L(W)$ that quantifies how well the network is currently doing on D, i.e. how close $F(x_j)$ is to y_j for all j. $L(W)$ is then minimized to find the best possible set of network weights W^*. This process is typically referred to as *training* the network and D is the *training dataset*.

To make this possible, the network $F(x)$ as well as the loss function $L(W)$ (which also includes the network) are selected so that they are differentiable. This means, W can be optimized via *gradient descent* (or a more advanced variant like for instance the popular Adam optimizer [KB15]), i.e. by taking small α-steps in the direction of $-\frac{\partial L}{\partial W}$, where α is the *learning rate*. Computing the full gradient, i.e. with respect to the entire dataset D (which may contain millions of samples), is generally far too slow. Therefore, it is usually only approximated in each step for a small random subset, a *mini-batch*, of the training data. While optimizing millions of parameters via small local updates would intuitively seem extremely difficult, it tends to work remarkably well in practice leading to all kinds of powerful neural network models. It should be noted that the actual goal is to perform well, i.e. output high quality (according to some measure) *predictions* $\hat{y}$, for new examples not contained in D but just drawn from the same distribution. If the training dataset D is large and representative enough, this *generalization* can usually be observed, which is one of the main reasons why deep learning is so effective.

So far, we have looked at neural networks only on a very high level, but what do the variables X, Y and f_i actually represent in practice? The inputs X can take many different forms, be it images [KSH12], speech recordings [AAA+16] or text [VSP+17] (all in appropriate numeric encoding). The same is generally also true for outputs, although a few specific types are particularly widespread. One of those are *classes*, i.e. (mutually exclusive) categories describing the input. The goal of the neural network is then to *classify* the given inputs, i.e. assigning to them the correct class. This could for example be determining the species of an animal depicted in an image. The outputs y_j in the training datasets are usually referred to as the *ground truth labels*, which are, in most cases, determined by humans. This thesis primarily deals with *object detection networks*,

which a) may produce multiple outputs for a single image and b) also estimate *bounding boxes* (the minimum area rectangle enclosing an object) in addition to object classes.

The final point we want to discuss in this section is what kind of functions f_i network layers typically are. The most basic type of layers are affine transforms followed by some non-linearity (without these, the layer-structure would be redundant as the whole network would collapse to a single affine transform). (2.2) shows an example of such a *fully connected* layer with a ReLU (Rectified Linear Unit) activation function (A is a matrix and b a vector).

$$y = \max\{Ax + b, 0\} \tag{2.2}$$

In the context of images, the most important type of f_i's is a *convolutional* layer. Such a layer has an $n \times c \times s \times s$ weight tensor, where n is the number of convolutional filters, c the number of input channels and s the size of each filter. Applying such a layer to a $c \times w \times h$ input (i.e. c channels, width w and height h) yields an $n \times w \times h$ output tensor. Each of the n filters is applied to every spatial (i.e. $w \times h$) location in the input and operates on the values of all c channels in an $s \times s$ neighborhood, eventually resulting in a $w \times h$ output. This process is illustrated in Figure 2.3. The two-dimensional outputs are also often called *feature maps* as they tend to encode the presence of certain "features" (e.g. a nose or an ear) in different parts of the image, especially in deeper network layers. A neural network with several convolutional layers is typically called a *convolutional neural network* or CNN for short. Convolutional layers are so effective on images that nowadays essentially all (including those subsequently discussed throughout this thesis) state-of-the-art vision related neural networks are CNNs.

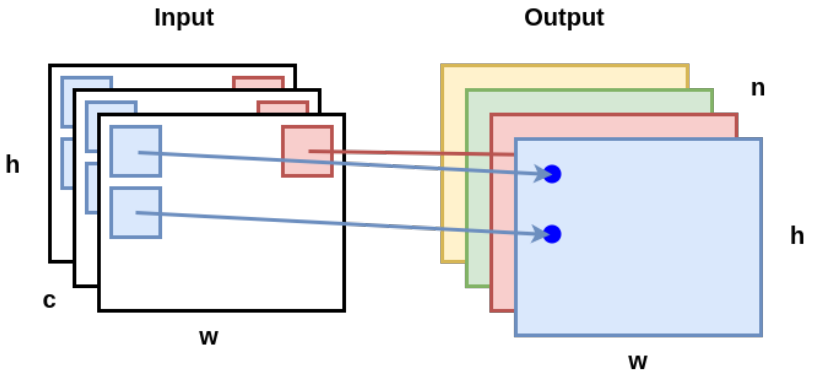


Figure 2.3: Visualization of a convolutional layer. An input with dimensions $c \times h \times w$ is transformed into an $n \times h \times w$ output through n convolutional filters applied to every spatial location of the input.

2.2.2 Evaluating Object Detection Quality

Determining how well a classifier does in distinguishing between cats and dogs is pretty simple. Counting how often it is right and how often it is wrong usually already gives a solid idea about how well a classifier performs. In contrast, it is much harder to formalize when an object detection model produces high quality predictions.

The first problem that arises in assessing the prediction quality of object detection models is quantifying how "well" two bounding boxes match. A good metric should yield 1 when two bounding boxes match perfectly and 0 when they do not overlap at all. However, it should be noted that a big bounding box fully enclosing a very small one is not a good match, hence simply computing the area of overlap is not sufficient. The generally accepted solution is the *Intersection over Union* or IoU value, which is computed according to (2.3), where A and B are bounding boxes, $|\cdot|$ denotes the area, $\cap$ the intersection and $\cup$ the union.

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

The next question that arises is which predicted bounding box should be matched to which ground truth box before calculating the IoU. A common and sensible approach is to first group both labels (i.e. the ground-truth box annotations determined by a human) as well as predictions (the boxes output by the model) per object class and then start matching the predictions in order of decreasing confidence (detection models usually return some measure of confidence for every detection) to the unmatched ground truth box with the highest IoU. This makes sense as we would naturally expect that the more confident predictions are also more accurate. After discarding all predictions with confidence $< c$ and (if applicable) IoU to its match $< u$ (predicted boxes are matched to ground truth ones as explained above) where c and u are some predefined thresholds, we can then count the number of matched predictions tp, of unmatched predictions fp and of unmatched labels fn. Figure 2.4 shows the result of a label-prediction matching for an example image, where boxes of the same color were matched together. The orange label has not been detected and is hence a false negative whereas the black prediction has no corresponding label and is a false positive. The tp, fp and fn values allow computing the standard *precision* and *recall* metrics as given by (2.4).

$$\mathit{precision} = \frac{tp}{tp + fp}, \quad \mathit{recall} = \frac{tp}{tp + fn} \tag{2.4}$$

Precision indicates the fraction of detections that are actual objects (of a certain type), whereas recall tells us the fraction of actual objects that are detected by our model. Note that both values change as the confidence threshold c is varied, where smaller c typically means higher recall but lower precision and vice-versa. The visualization of this relationship is also known as the precision-recall curve or short pr-curve (see Figure

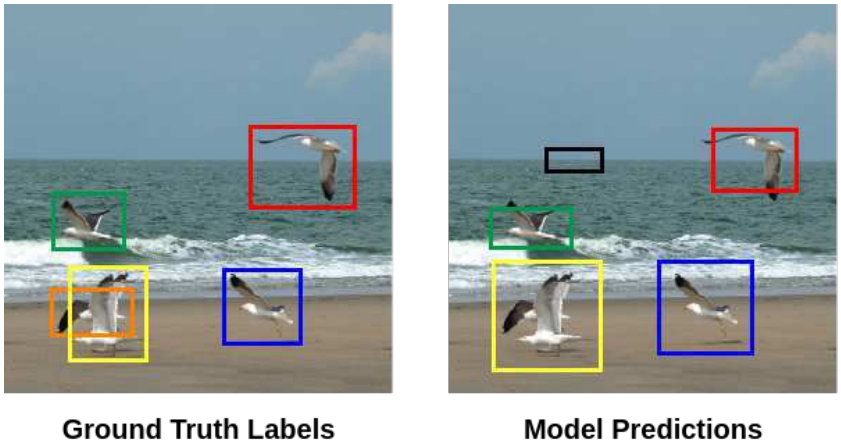


Figure 2.4: Example result of a label-prediction matching. Background image from MS-COCO [LMB+14], labels and predictions hand-drawn for demonstration purposes.

2.5 for an example). Since c is model as well as application dependent and because comparisons between models are much easier with just a single performance number, one usually computes the *average precision* or AP, which corresponds to the area under the pr-curve. An ideal model has 100% precision and 100% recall and thereby also an AP of 1. While this is certainly a good summary metric, it is a bit hard to interpret in terms of real observed model performance. Thus, it is advisable to also look at specific points on the pr-curve when doing practical work. The mean over all APs (which are computed class-wise) is known as the mAP, which is the main metric in most standardized object detection benchmarks such as Microsoft COCO [LMB+14]. There is, however, still one variable left, the IoU-threshold u. Earlier works used to fix this variable to 0.5 indicating reasonable localization accuracy (with the respective mAP@.5). However, over the years object detection models have become so powerful that a new evaluation metric was proposed, the mean over mAP@u for $u \in \{.5, .55, \ldots, .95\}$ called the mAP@.5:.95. Since high mAPs at large values of u require extremely precise localization, this new metric is a lot harder to do well in. It also needs to be noted that the mAP@.5:.95 is primarily of academic interest as it aggregates so many different scenarios that its practical performance becomes hard to judge.

Now that we have addressed the proper evaluation of object detection systems, we can finally move on to discussing how such systems work.

2.2.3 Method Families

Broadly speaking, most current object detection techniques can be categorized as a member of one of two families, *one-stage* methods and *two-stage* methods [JZL+19].

Two-stage methods first compute a number of image regions that are likely to contain an object. Those are then cut out of the image and individually run through a CNN

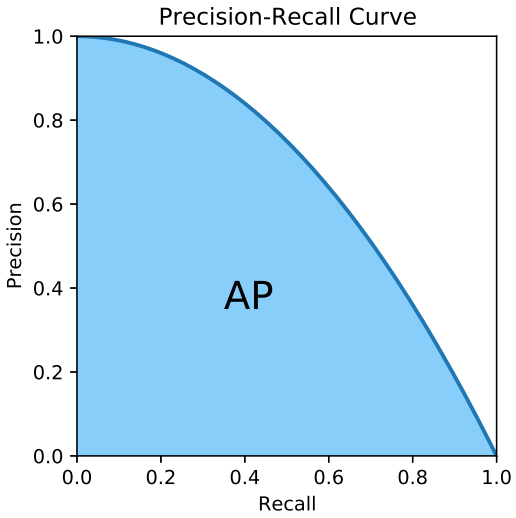


Figure 2.5: Example of a precision-recall curve, the corresponding average precision is the area shaded in light blue.

which determines a) if they indeed contain an object and b) if they do, its class as well as a refinement of the initial bounding box given by the crop-out. On the other hand, one-stage methods predict bounding boxes and object types simultaneously in just a single neural network pass. This typically takes the form of outputting a dense grid with bounding boxes, confidence scores and object classes for every grid-cell.

Historically, two-stage algorithms were considerably more accurate than one-stages methods as their predictions are not bound to any grid, but also much slower since they require a large number of CNN evaluations to process just a single image [GDDM14, RDGF16]. While the speed difference is still there nowadays, several recent innovations have allowed one-stage methods to essentially close the accuracy gap, at some points even surpassing existing two-stage detectors [LGG+17]. Current state-of-art one-stage detectors can easily achieve > 30 FPS on consumer graphic cards [RF18, BWL20], which makes them the clearly superior choice for real-time applications, such as the work in this thesis.

We will now look at the major techniques, optimizations and other best practices powering state-of-the-art variants of both types of detectors in the next several sections.

2.2.4 Region Proposals and One-Stage Detectors

As already mentioned earlier, two-stage methods initially determine “interesting” image regions before cutting them out and passing them on to a classification network. The very first models like R-CNN [GDDM14] accomplished this via classical computer vision methods like selective search [UVDSGS13]. However, these classical methods were quickly

replaced by a much more accurate region proposal network, or RPN for short, introduced in Faster R-CNN [RHGS15]. Since fewer incorrect region proposals directly translate to fewer unnecessary classification network runs, this further made the whole system a lot faster.

But how can one design a neural network that predicts not just a single set of class probabilities but a large number of different regions? Intermediate tensors of a CNN correspond to spatial maps of feature vectors associated to certain parts of the input image, also known as their receptive field. As the input passes through deeper network layers, the spatial resolution of the feature maps decreases while the size of the receptive fields increases. In most classification architectures, the spatial resolution eventually becomes 1×1, meaning that the one remaining feature vector was computed by considering the entire image. Faster R-CNN's RPN runs the input through a sequence of convolutional layers eventually resulting in a rich feature map with dimensions $c \times h \times w$ where c denotes the number channels, h the height and w the width. It then applies another set of convolutional filters, which correspond to sliding a small window over all $h \times w$ spatial locations, to get a dense $(1 + 4) \times h \times w$ grid with 1 objectness score (an estimate of how likely it is that the cell contains an object) and a bounding box prediction consisting of 4 values (x, y, w, h) for every location (actually the RPN does a bit more than that as we will discuss shortly). The latter is relative to the cell in the grid such that it can be computed with the same weights for every location. During training, objectness targets are assigned to grid-cells that have high overlap (in the original image) with a labeled object bounding box.

Early one-stage detectors like YOLO [RDGF16] or SSD [LAE+16] had basically the same general structure as the RPN described in the previous paragraph, they just also output an additional object class for every point in the grid. Making such a network really effective in practice required several further enhancements, which we will discuss shortly. See also Figure 2.6, which depicts the general structure of an RPN and a one-stage detector. Although most of those enhancements were also employed by Faster R-CNN's RPN, we think they are easier to explain in the context of actual detectors, i.e. one-stage models.

The first obvious problem with the approach outlined thus far is that a single grid-cell might actually contain more than one object. This is especially problematic when two objects of different classes overlap as the model can only predict a single class per cell. Second, different types of objects have very different shapes of bounding boxes, just compare the tall but slim box surrounding a standing person with the square box encapsulating a ball. This makes it difficult to learn a single model that gives precise bounding box estimates for all objects types. A key technique that yields significant improvements in both outlined areas are *anchor boxes* [RF17]. Rather than returning just a single box per location, anchor based detectors make multiple predictions relative to several default boxes with varying aspect ratios and scales, again centered in each grid-cell. In this context "relative" means that the network outputs offsets $(\Delta x, \Delta y, \Delta w, \Delta h)$ to the anchor rather than the coordinates directly. As an additional benefit, anchor boxes

also help with training as initial predictions at the start are already much closer to the labeled bounding boxes. A widespread way of determining the anchors is to run k-means clustering for $5 \leq k \leq 9$ with IoU-distance defined by $1 - \text{IoU}(\cdot)$ on the boxes in the training dataset as is done for instance by YOLOv2 [RF17]. k-means clustering tries to find a set of k prototype boxes such that the average IoU-distance of a training box to its closest prototype is as small as possible (hence the IoU-distance rather than the normal IoU is used), making the determined prototypes a natural choice for good anchors.

Lastly, there is another important technique deployed by essentially all object detectors, that is *non maximum suppression* or NMS for short [GDDM14]. Bigger objects have significant overlap with multiple grid-cells and anchors. Thus, assuming a reasonably smooth model, there will usually be many different predictions of the same object within and across adjacent grid-cells. However, their confidence will vary depending on how good of a match they are. Hence, an effective strategy to get rid of redundant predictions is to eliminate boxes with high IoU-overlap but low confidence. More precisely, let $\mathcal{P}$ be the set of predictions returned by the model, then the result $\mathcal{P}'$ of NMS is given by (2.5), where t is an IoU-threshold, typically set around $\frac{1}{2}$. Note also that by using the IoU we will not accidentally filter away overlapping objects with very different shapes. As a final remark, performing NMS is an inherently quadratic computation and thereby actually quite expensive, which is why it is usually only computed for boxes that have at least a certain confidence (of which there are typically not that many).

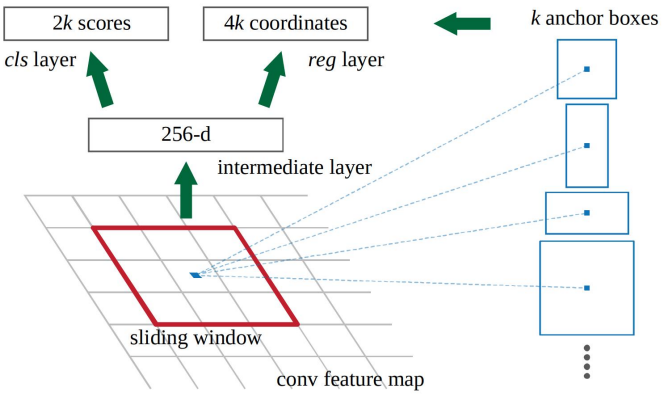
$$\mathcal{P}' = \{p \in \mathcal{P} \mid \nexists\, p' \in \mathcal{P} : \text{iou}(p, p') > t, \text{conf}(p') > \text{conf}(p)\} \tag{2.5}$$

This concludes the discussion of the most fundamental techniques underlying the majority of current single-stage object detectors and similarly region proposal networks. We will now continue elaborating on several more advanced ideas for speeding up the inference process as well as improving general detection quality.

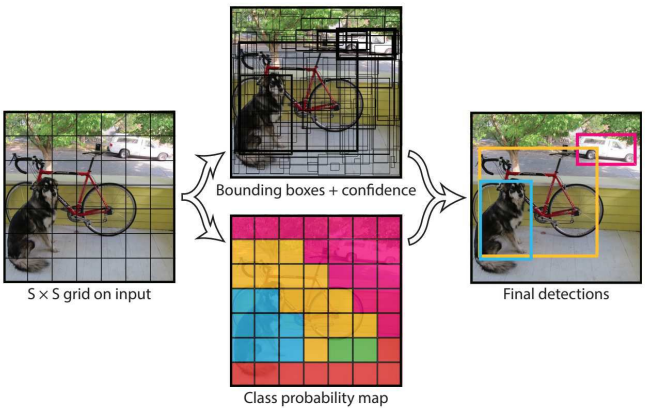
2.2.5 Speeding Up Two-Stage Methods

The RPN mentioned in the previous section was just one of several improvements that led from the original R-CNN over Fast R-CNN [Gir15] to eventually Faster R-CNN [RHGS15]. We will now focus on two more of those improvements as they are great demonstrations of recurring themes which have led to significant speed-ups in various types of computer vision models. In short, those improvements are pooling differently sized regions to a common shape for better batch processing and sharing weights/features between different sub-networks.

One of the primary reasons why R-CNN took multiple seconds to process a single image was that it had to individually run a CNN for hundreds of small image crop-outs. Fast R-CNN instead computes a big convolutional feature map for the entire image (similar to the RPN described in Section 2.2.4) on which the region of interest selection is then performed. This is accomplished by *Region of Interest pooling* (or RoI-pooling) as



(a) RPN, figure from [RHGS15].



(b) One-stage detector, figure from [RDGF16].

Figure 2.6: General structure of an RPN and the work-flow of a one-stage detector.

introduced in [HZRS15], where a rectangular $h \times w$ region is reduced to a smaller square of size $s \times s$ by aggregating the feature vectors of $h/s \times w/s$ tiles each. Small network heads directly follow up with the final predictions. Since many region proposals usually overlap, this whole scheme avoids large amounts of duplicate feature computations and thereby improves inference time by orders of magnitude.

However, when the region proposals are computed by an individual RPN, this still means that detection requires two expensive deep CNN runs. Here Faster R-CNN employs another (nowadays) common trick, namely sharing the majority of features between the RPN and the detection network. More precisely, a small network head computes region proposals directly from the same big feature map that is then also RoI-pooled. Since computing this map, which we would need for detection anyway, completely dominates the computational work of proposing RoIs, the region proposals are almost free. This is also one of the main reasons for further significant speed-ups from Fast R-CNN to Faster R-CNN.

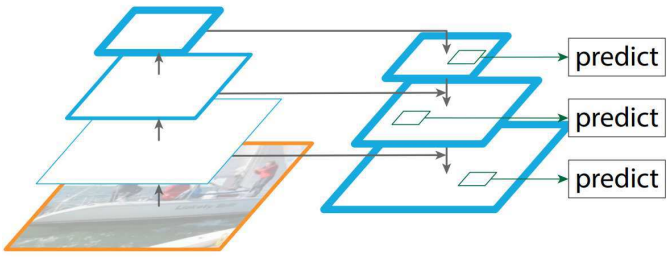
It needs to be noted that these optimizations are usually very effective but in general not equivalent to the initial formulations. In theory, a fully separate RPN might be able to learn more specialized features that are exclusively useful for accurate region proposing but not for detection. However, in practice one can often observe the opposite effect, namely that sharing features provides stronger regularization, makes training easier and frequently even leads to more robust models (e.g. compare [Gir15] and [RHGS15]). Hence, these type of tricks are widely used to great effect.

2.2.6 Feature Fusion

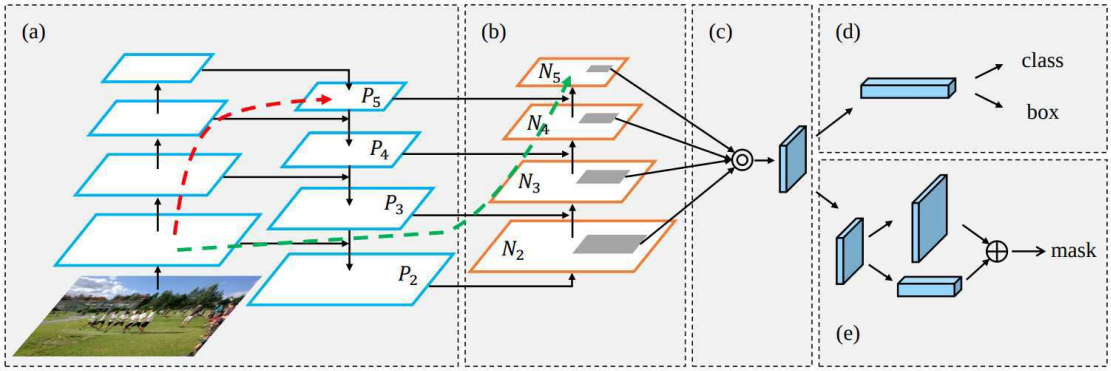
Another big challenge is that the objects to be detected often have very different scales, e.g. there might be a horse right in front of the camera and a bird somewhere in sky, which we would both want to detect. Looking back at the approaches discussed in Section 2.2.4, the horse spans several grid-cells whereas the bird only occupies a tiny fraction of a single cell. Since CNNs typically lose spatial resolution as the input passes through deeper layers, it will be very difficult to maintain the precise position of the bird. Increasing the grid-size on the other hand would make it problematic to detect the large horse.

Feature Pyramid Networks (FPNs) [LDG+17] attempt to address this dilemma by making predictions on multiple grids of different sizes. The most obvious way to do this would be via several networks for different image resolutions, which would, however, be prohibitively inefficient. Instead, FPNs directly exploit the structure of typical CNNs. That is they upsample (e.g. by linear interpolation) the feature map of the final layer and then combine it with the one of an earlier layer that has twice the resolution. This process can be repeated recursively to yield additional pyramid levels. One intuition behind passing up the higher level features is to bring additional context into the more fine-grained feature maps, for example, it may be easier to detect a bird when you know that it is surrounded by the sky. Eventually separate predictions are output for every pyramid level before being merged back together via NMS.

Since FPNs have been proven to be very effective, a lot of research has gone into developing even better feature fusion methods. BiFPN [ZDH+18], for instance, introduces an additional downwards pathway, i.e. where it starts with an earlier layer and iteratively refines it with higher and higher upsamplings of the smaller and smaller later layers. Eventually, the result is merged with the final (standard) FPN feature map. Additionally, different feature maps are not just combined by trivial element-wise addition but the merging actually happens through small convolutional modules, which allows the network to fuse the features much more intelligently. Alternatively, PANet [LQQ+18] computes the top-down path not from the original network layers but from the output of the bottom-up feature fusion (see also Figure 2.7). Further, the authors pool together all maps of the top-down path rather than computing individual predictions for each of them. Very recently, EfficientDet [TPL19] takes it one step further and performs a neural architecture search to find a particularly effective (albeit not very intuitive) feature fusion scheme. This is still a very active area of research and further improvements are to be expected.



(a) FPN, figure from [LDG+17]



(b) PANet, figure from [LQQ+18]

Figure 2.7: Visualizations of different feature fusion architectures.

2.2.7 Special Loss Functions

Strong object detection models are not at all straight-forward to train as the neural networks have to strike a balance between several different tasks (e.g. estimating objectness, regressing bounding boxes and classifying objects). Other issues arise due to things like foreground background imbalance or initial bounding box predictions being completely off the targets. Particularly advancements in the design of special loss functions have helped to make training detectors a lot easier. We will now outline two very popular techniques, *focal loss* [LGG+17] and *generalized IoU* [RTG+19] (or gIoU).

Most one-stage detectors output dense predictions for multiple anchor boxes over several grids of different granularity. However, since most images contain only a handful of objects, almost all of the targets during training will be background samples. This means becoming a tiny bit more confident on the huge number of very clear background samples will most likely decrease the loss more than becoming a lot more confident on the much harder but also much rarer actual objects. This not only slows down training considerably but can even result in less accurate models in the worst case.

As a solution Lin et al. propose the focal loss function $FL(p_t)$ defined by (2.6) where p_t is the predicted score of the correct class for target t and λ is a hyper-parameter (experiments indicate that $\lambda = 2$ is a good choice). The factor $(1 - p_t)^\lambda$ exponentially

downweights examples that are already well classified with $p_t \gg 0$ while not affecting the loss much for ones that are still hard with $p_t \approx 0$. The term $-\log(p_t)$ carries over from the standard cross-entropy.

$$FL(p_t) = -(1 - p_t)^{\lambda} \log(p_t) \tag{2.6}$$

This quite simple, yet very powerful trick allows the authors to train RetinaNet, the first one-stage detector to surpass all (at that time) two-stage methods. For that reason focal loss quickly became a widespread technique that can be found in many more recent object detection works.

The box regression part of object detection models is usually trained on absolute or quadratic differences between the predicted and the ground truth coordinates. Evaluation is however usually based on the IoU as elaborated in Section 2.2.2. Ideally, one would directly optimize for the metric that matters, however the IoU is 0 when there is no overlap between prediction and target, hence there would also not be any gradient to learn from. The gIoU attempts to address this problem. Let C be the smallest convex shape enclosing bounding boxes A and B, then the gIoU is given by (2.7). In words, the gIoU corrects the standard IoU downwards by the fraction of the area of the enclosing shape the two boxes do not occupy (which is directly related to how far apart they are).

$$\text{gIoU}(A, B) = \text{IoU}(A, B) - \frac{|C \setminus (A \cup B)|}{|C|} \tag{2.7}$$

C can be shown to always be a rectangle that is computable from the coordinates of A and B via a few min-max operations. As a consequence, it always has a gradient even when the boxes do not overlap and can therefore directly be used as an optimization objective. Experiments demonstrate consistent improvements in the final model performance when training with gIoU over a standard box regression loss [RTG+19].

2.2.8 Training Tricks

In addition to the special loss functions described in Section 2.2.7, state-of-the-art detectors also utilize many other training tricks, which is the topic of this part.

First, detection models are rarely trained completely from scratch. Most one-stage architectures consist of a deep convolutional network backbone intended to produce a rich feature map, from which in succession small (e.g. just a few convolutional layers) network heads regress bounding boxes and predict object classes. Typically, this backbone (plus a simple single class prediction head) is first pretrained on a huge classification dataset such as ImageNet before the actual object detection training (with the corresponding detection heads) starts. This greatly helps with training since the classification features are usually already a solid starting point for detection. Hence, further learning can set focus on the much smaller object detection heads while just refining the millions of parameters of the

earlier layers. Additionally, available pure classification datasets are orders of magnitude bigger than detection ones and CNNs with just a classification output are empirically a lot easier to train. In fact, many recent works directly use previous classification models as a starting point, EfficientDet is for example based on EfficientNet [TL19].

Data augmentations, i.e. virtually increasing the size of a dataset by intelligently transforming existing training images, is omnipresent in machine learning, especially in computer vision where labeling new images is a very time consuming task. Complementary to the standard horizontal flipping, resizing, color shifting and distortion augmentations, there are also a few other interesting ideas found primarily in the object detection literature. One of those is *mosaic augmentation* where parts of four different training images are stitched together to form a continuous mosaic. This works by lining up four images with half their original resolution (so that more of them will be visible) in a 2×2 grid followed up by cutting out a new image around the grid's center point, see Figure 2.8 for a few examples. In addition to forcing the detector to learn recognizing objects in unusual contexts, using mosaics is also an efficient way to learn from the lower resolution versions of the training images, which in turn tends to improve the detection of smaller objects at inference time.

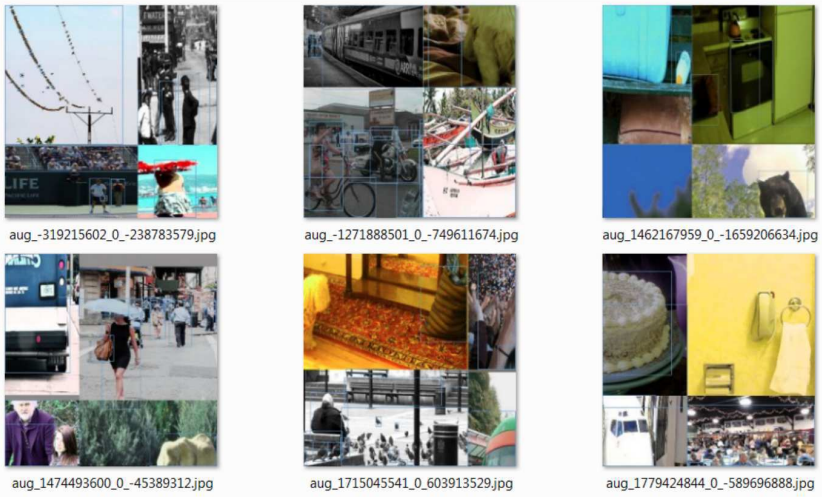


Figure 2.8: Mosaic data augmentation examples, figure from [BWL20].

Other interesting training techniques include optimizing the choice of model hyper-parameters (like the learning rate of the optimizer or the amount of a weight decay penalty) via genetic programming. For that purpose the model is trained a few epochs for several candidate configurations, the best ones survive and are then mutated before the whole process starts again. Another new strategy is *self-adversarial training* (SAT) where adversarial examples, e.g. slightly altered training images that the current model confidently rates as containing no objects, are added to the training set. These are computed optimizing not the CNN weights but the input image.

While this section covers several of the most interesting and relevant training tricks for object detection models, this list is by no means exhaustive. An in-depth study about many more optimizations can be found in [BWL20].

2.2.9 State-of-the-art Performance

We will conclude our treatment of modern object detection methods by briefly talking about how well they actually work.

Perhaps the most fundamental object detection benchmark is the Microsoft COCO [LMB+14] dataset. It consists of images containing a total of 80 different object classes, ranging from animals to humans and machines. Table 2.1 lists the performance of selected object detection methods on the COCO test-set. A detailed explanation of the evaluation metrics can be found in Section 2.2.2.

Model	**mAP@.5 [%]**	**mAP@.5:.95 [%]**	**FPS (M/P/V)**
YOLOv3 @ 416×	55.3	31.0	35/-/-
YOLOv4 @ 416×	62.8	41.2	38/54/96
YOLOv4 @ 512×	64.9	43.0	31/43/83
YOLOv4 @ 608×	65.7	43.5	23/33/62
EfficientDet-D2	62.3	43.0	-/-/41.7
TridentNet DCN	67.6	46.8	-/1.3/-
Cascade R-CNN	62.1	42.8	-/8/-

Table 2.1: Performance of several object detection models on the Microsoft COCO test-set. M/P/V represents the FPS achieved on NVIDIA GPUs with Maxwell, Pascal and Volta architecture, respectively. All values were taken from the comparison in [BWL20].

The performance gap between YOLOv3 and YOLOv4 shows that utilizing many small tricks, such as those in Section 2.2.8, can lead to significant gains even when the general architecture stays mostly the same. Next, we can observe that increasing the test-time resolution (while the model does not change) typically yields slightly better results but comes at the cost of a significant drop in inference speed. We also see that the state-of-the-art single-stage methods YOLOv4 and EfficientDet outperform the advanced R-CNN variant Cascade R-CNN [CV18] in terms of accuracy while running in real-time (at least on fairly strong graphic cards). Only the extremely expensive TridentNet [LCWZ19] (DCN type) manages to be slightly more accurate in exchange for evaluating barely more than one image per second. In general, mAP@.5s of ≈ 0.6 and mAP@.5:.95s of ≈ 0.4 are already very good scores (even though 1.0 is the theoretical optimum) as visual inspection on a few samples prediction confirms.

In the last few months, there have also been a few detection competitions for traffic datasets. The Waymo 2D-Detection Challenge[1] was won by an ensemble of a large number

[1] `https://waymo.com/open/challenges/2d-detection`, accessed: 2020-07-17

of different models within the SimpleDet framework [CHL+19]. This solution achieved 0.79 mAP on L1- (i.e. rated easy) and 0.74 on L2-objects. These scores being quite a bit higher than the best mAP@.5 on COCO is probably due to prediction specialized for traffic settings being slightly easier than general purpose object detection. Meanwhile, the best model operating exclusively on camera data in the nuScenes[2] 3D-detection challenge was CenterNet [ZWK19] with an mAP of 0.338. All in all, these results suggest that current state-of-the-art detection models are already ready (or at least very close to being ready) for real-world use in driving situations. However, it needs to be noted that both the Waymo as well as the nuScenes dataset primarily consist of daytime data recorded in cities and not nighttime images in rural areas, which this thesis is particularly interested in.

2.3 Multiple Object Tracking

So far, we have been primarily concerned with locating multiple objects in a single frame. However, in many situations it is not only crucial to know where certain objects are at the current point in time but also how they got there. Is the car we see now the same we saw one second ago or are there now actually two cars, one of which we currently cannot see for whatever reason? Correctly answering questions like these is critical to make correct driving decisions (e.g. whether we can safely change lanes and pass the car right in front of us). Also, as good as modern object detection systems already are, they still make mistakes, sometimes even extremely obvious ones. Using information about prior detections can be helpful to smooth out those errors, since (at least intuitively) it should be much easier to redetect objects if their particular shape and approximate location is already known from the previous frame.

This fundamental extension of object detection is generally referred to as *multiple object tracking* or MOT. More formally, the goal of MOT is to a) detect multiple objects (optionally determine their type) and b) assign to them identifiers so that the same (physical) object instance has the same ID in every frame it is detected (ideally in every frame it appears).

This section covers state-of-the-art object tracking methods. It begins with a discussion about how to properly judge the performance of good trackers, a surprisingly challenging task. Then follows an overview, also including advanced ideas, about a popular framework for directly extending state-of-the-art object detection methods into full trackers. The recently emerging pure neural network trackers will be mentioned as well. Lastly, a short summary of the currently best MOT systems will be presented.

2.3.1 Evaluating Tracking Quality

While the quality of detections was already rather not straight-forward to assess (see Section 2.2.2), it is even more difficult for the task of tracking. How “well” is a system

[2] `https://www.nuscenes.org/object-detection`, accessed: 2020-07-17

doing that accurately tracks an object for some time, then drifts off a bit, but eventually manages to get on track again with a different object ID? How can this performance be quantified in a comparable manner? Obviously, it will be very hard to capture all characteristics of a good tracking system in just a single number. Over the years, the CLEAR-MOT metrics [BS08] have become the de-facto evaluation standard in the tracking literature.

Similar to object detection, the first step of computing CLEAR-MOT metrics is to match all predictions for a frame to the most appropriate labels. However, simply assigning every predicted box to the labeled box with highest IoU (if it exceeds a certain threshold u) is not a good strategy as objects are expected to cross paths often and may thus heavily overlap for a few frames. Hence, the matching must be very careful not to introduce any errors the system is not actually making (e.g. switching the IDs of overlapping tracks). This and other phenomena make proper association of predicted and labeled tracks rather complicated, which prompted the development of standardized evaluation frameworks such as py-motmetrics[3]. Please refer there for detailed descriptions of the matching process; we will now simply assume that appropriate matches have been found.

With the matches in place, we can already compute several standard metrics such as precision, recall, false positive or false negative rate. These provide good insights into the general detection rates of the system under inspection. The first tracking specific score is the number of *ID-switches* giving us information about how consistently the same ID is assigned to a track of a single object. For frame t let g_t be the number of ground truth objects, fp_t be the number of false positives, fn_t the number of false negatives and ids_t the number of incorrectly assigned object IDs. Then we can define the *multiple object tracking accuracy* (or MOTA for short) as follows in (2.8).

$$MOTA = 1 - \frac{\sum_t (fp_t + fn_t + ids_t)}{\sum_t g_t} \tag{2.8}$$

The MOTA summarizes all types of tracking errors by relating them to the total number of ground truth labels. Note that while the MOTA is capped at 1, it can also easily be negative if the system is making a large number of mistakes. Since the MOTA considers every single tracking mistake, it is usually hard to achieve scores near the maximum. The MOTA however does not take into account how precisely (e.g. in terms of box overlap) objects are actually tracked (as long as the IoU is below u). For that purpose there exists the *multiple object tracking precision* or MOTP. It is defined by (2.9), where m_t is the total number of matches and d_t^i the distance of the match i to its corresponding ground truth object.

$$MOTP = \frac{\sum_{t,i} d_t^i}{\sum_t m_t} \tag{2.9}$$

[3] `https://github.com/cheind/py-motmetrics`, accessed: 2020-07-18

Intuitively, the MOTP measures the average distance (in terms of $1 - \text{IoU}(\cdot)$) between tracked objects and their ground truth. In contrast to the MOTA, its best possible value is 0. Note that this does not factor in the tracking accuracy at all, e.g. a tracker that only detects an object in a single frame but perfectly predicts the bounding box would have perfect MOTP but very low MOTA. This demonstrates that it is extremely important to always look at various different metrics when evaluating the performance of a tracking algorithm.

2.3.2 Tracking by Detection

The most common way to extend the powerful CNN-based object detectors to multiple object trackers is the *tracking by detection framework*. The main idea is to compute independent detections for every frame and then match them up to those in the previous frames. This approach is particularly appealing since new advances in detection techniques usually directly translate over to tracking improvements. Additionally, the framework decomposes the problem of tracking into multiple sub-problems that can (at least to some extent) be tackled individually. Before we begin to study more advanced techniques, we will first take a look at probably the most straight-forward, yet often already very effective, MOT baseline known as SORT [BGO+16].

The SORT method operates exclusively on bounding boxes (the significance of this will become clear in Section 2.3.3). Current tracks are modeled by Kalman filters which not only try to smooth out the trajectories but also attempt to estimate object velocities and thereby predict their locations in the next frame via assumption of linear motion. Note that this happens on the level of bounding boxes. In addition to modeling the change in location by the velocities v_x and v_y, there is also a scale-velocity v_s that allows the box to increase in size over time (the aspect ratio remains fixed for simplicity). The neural network detections in the next frame are matched up with the Kalman filter predictions of the existing tracks. This happens by building a bipartite graph with edges between predicted box p_i and existing track t_j of weight $w_{ij} = \text{iou}(p_i, t_j)$ if $w_{ij} > u$ for some IoU-threshold u. The maximum weight matching on this graph, computed for example with the classic Hungarian method, gives the assignment of predictions to tracks where unmatched predictions just spawn new tracks and existing tracks are deleted if they remain unmatched for more than a few frames. Note that the Kalman filters are especially important in low frame rate regimes where the overlap of an object across two frames is so small that it would not reliably be matched to the same track without a motion model.

Although SORT is a rather simple procedure, it already works surprisingly well in most scenarios. This is primarily due to the fact that objects typically only move rather little from one frame to the next and are thus fairly easy to match. Problems mostly occur in difficult situations where different objects obstruct each other or when one object disappears for a longer period of time. These situations are comparatively rare in most datasets (and thus even simple SORT often already reaches high MOTA scores), however they might actually be exactly those where having reliable tracking information is most

critical. Hence, many much more advanced tracking techniques have been proposed over the years, some of which we will look at throughout the next few sections.

2.3.3 Re-identification Features

Certainly the biggest shortcoming of SORT is that the matching operates exclusively on bounding boxes. If there are two similar bounding boxes in roughly the same location, SORT will have a very hard time correctly continuing the tracks. This is true even when the corresponding objects are clearly different, just think of two pedestrians, one wearing a red and the other a blue shirt. Most tracking systems address this issue via so called *re-identification features*. Typically, those come in form of additional feature vectors output for each instance by a CNN and their similarity is subsequently taken into account when matching new detections to existing tracks.

One of the first attempts at incorporating such additional appearance features into the matching procedure was DeepSORT [WBP17]. Similar to two-stage object detection methods, the authors compute feature vectors by applying a rather small additional CNN to the detection (in the original image). This network was pretrained on a large person identification dataset. Weight w_{ij} on the assignment graph is then computed as a weighted combination of the box distance (the actual method described in [WBP17] is a bit more complex, but that is not important for the current discussion) and the cosine-distance $p_i^\intercal t_j$ of the prediction- and the track-features denoted by p_i and t_j respectively. Using this new techniques, they report an almost 50% reduction of ID-switches for challenging MOT datasets. More recent works like [WZLW19] attempt to share neural network layers between the object detector and the appearance feature generator in a similar fashion to how Faster R-CNN computes region proposals almost for free (see Section 2.2.5). This allows employing much deeper networks computing richer representations without dramatic loss in speed.

Another interesting topic is the way re-identification feature generators are being trained. The key challenge here is that the feature vectors v_i and v_j should be close if they capture the same object instance at different points in time but very different if they do not. Additionally, this should also be the case for new instances not already contained in the training set. There is no guarantee that the features in the last layer of a classification network have such properties, hence special training techniques have been developed. One of the most successful is training with *triplet loss* [SKP15]. The main idea of this special loss function is to minimize the (Euclidean) distance of the d-dimensional feature embedding of sample x_i denoted by $f(x_i) \in \mathbb{R}^d$ to the embeddings of other positive samples $f(x_{p(i)})$, i.e. ones of the same identity, while maximizing it to the embeddings of negative ones $f(x_{n(i)})$. More precisely, we want $f(x_{p(i)})$ to be closer to $f(x_i)$ than $f(x_{n(i)})$ by at least the value of α (given that all embeddings are normalized to have unit length). Then the full optimization criterion can be written as (2.10), where the summation happens over all relevant triples $\mathcal{T}$.

$$\sum_{i,p(i),n(i)\in\mathcal{T}} \Big[||f(x_i) - f(x_{p(i)})||_2^2 - ||f(x_i) - f(x_{n(i)})||_2^2 + \alpha\Big]_+ \tag{2.10}$$

Always considering all triples would be extremely inefficient as this would require evaluating the full network for the whole dataset at every learning step (and most trivially fulfill the margin condition anyways). Instead the authors suggest to focus only on the hardest triples, i.e. ones where the constraint is maximally violated. Further, those are approximated by selecting the $f(x_{p(i)})$ with the maximum and the $f(x_{n(i)})$ with the minimum distance to $f(x_i)$ within a sufficiently large mini-batch. Overall, this training strategy has proven to be a very effective means to build powerful identification models.

2.3.4 Advanced Matching Schemes

Given the object locations in all frames, finding their best possible assignment to consistent tracks (according to some criterion) is a combinatorial optimization problem. Usually, and especially for hard instances, the global solution for such problems cannot be found by only making a sequence of locally optimal choices, e.g. always assigning objects to their best matching track at every timestep like SORT is doing. While this is certainly most relevant for offline tracking (where detections for all future frames are available), it is also very much worth considering for online methods. Imagine for instance a system that can retroactively correct wrong or very uncertain assignments once more information becomes available. At first glance this might seem like it could introduce highly dangerous (for an autonomous vehicle) sudden jumps (when an object changes identity from one frame to another), but remember that the entire tracking history would get corrected and a controller could thus simply look back in time and replan accordingly. We will now take a closer look at this highly challenging algorithmic problem and some techniques for addressing it.

In particular, we will now describe the NOMT algorithm introduced in [Cho15], which has led to good results on driving related tracking benchmarks. At every timestep t NOMT generates a set of track hypotheses for all detections in the last τ frames; the most consistent hypothesis set is then returned as the solution. To do this, the authors first formulate their problem in an energy minimization framework, which we will now sketch very loosely. Let D be the set of detections at the current timestep, H the set of selected hypotheses and h_d the hypothesis in H containing d. The structure of the energy function $E(H)$ is then given by (2.11).

$$E(H) = \sum_{d\in D} \psi(d, h_d) + \sum_{h_1,h_2\in H} \phi(h_1, h_2) \tag{2.11}$$

$\psi(\cdot)$ expresses the compatibility of a detection d with its corresponding track, which includes a term accounting for feature consistency between detections of the track and d, as well as a smoothness constraint. $\phi(\cdot)$ penalizes selected hypotheses with large

image overlap as well as duplicate detection assignments. As directly minimizing this energy function would be infeasible due to the exponential number of options for H coupled with the highly complex structure of the energy function (not visible in the simplified formulation of (2.11)), the authors heuristically generate several candidate hypotheses. Those include greedily built trajectories, matches of new detections to tracks from the previous timestep and new tracks starting with the most current set of predictions. Eventually, NOMT builds an undirected graphical model where the nodes are new detections and the states are hypothesis indices. Connections are added between nodes that compete for any mutually exclusive hypotheses. The value of a solution is given by energy function from (2.11). Since τ is typically not too large and the graph not that densely connected in most situations, the best hypothesis set can be determined in near real-time using a dedicated graphical model solver.

2.3.5 Pure Neural Network Trackers

Decomposing a big problem into several much smaller sub-problems that can be worked on more or less independently is one of the core principles of computer science. However, in areas that are particularly well suited for machine learning techniques it often turns out that a bigger model trained end-to-end significantly outperforms systems consisting of multiple individually tuned components. Following this trend, recent works on object tracking attempt to tackle the entire task directly "in one go" with just a single neural network rather than through separate detection, re-identification and matching phases. These types of models are also sometimes referred to as *pure neural network trackers*.

An early attempt at such a model was Tracktor [BMLT19], an extension to Faster R-CNN. Its main idea is to utilize the bounding box refinement component for realigning previous predictions (e.g. of existing tracks) in the current frame. This is accomplished by applying RoI-pooling on the features of the current frame not for the corresponding region proposals but for the tracked boxes from the previous frame. Since the subsequently applied prediction network is trained to precisely snap the coarse region proposal to the corresponding object instance, it can also quite reliably move the old box to the object's new location, at least provided that it has not moved too much between frames. Detected objects (from current region proposals) with little IoU to existing tracks start new trajectories. One big advantage of Tracktor is that it does not require any tracking specific training or data. Overall, the model performs pretty well in most usual situations, somewhat similar to SORT.

A very recent pure neural network tracker, which achieves state-of-the-art performance on various datasets, is CenterTrack [ZKK20]. Its basis is formed by CenterNet [ZWK19], a model that detects objects by their center points rather than with respect to anchor boxes. This is especially interesting for tracking purposes as it allows easily passing the previous frame's detections as an additional input taking the shape of a single 2D center heat-map. Such a heat-map is a rather fine grid in which only cells close to the center of an object have a non-zero value (see also the third image in Figure 2.9). Additionally, CenterTrack also sees the previous image when predicting the next center map. Objects

are identified as local peaks in the heat-map and simply matched greedily to the closest center of an existing track within a certain radius (while considering an offset that is also predicted for every heat-map location). An illustration of the CenterTrack framework can be found in Figure 2.9. Experiments show that this model is particularly effective in low frame rate regimes as the prior detections combined with the last image seem to help relocating objects even if they have moved significantly. Interestingly, it also turns out that heavy data augmentation makes it possible to train the model just on non-video data without dramatic loss in eventual tracking ability. However, CenterTrack is also not performing very well in really difficult tracking situations, for example it cannot really deal with situations where objects disappear for multiple frames only to reappear later or when two objects (with different shapes) almost share the same center point.

In summary, pure neural network trackers are a very interesting new research direction. Current approaches already show promising results and future breakthroughs seem quite likely.

Figure 2.9: CenterTrack framework; the offsets provide information of how an object has moved from the previous frame allowing more precise association. Figure from [ZKK20].

2.3.6 State-of-the-art Performance

To complete our overview of deep learning based object tracking techniques we will look at how well some of the best methods are actually performing on standard datasets.

Two of the best known MOT datasets are the MOT-Challenge 2017 [MLTR+16] (leaderboard found here[4]) about tracking pedestrians in crowded environments and the daytime vehicle tracking dataset KITTI [MG15] (leaderboard found here[5]). For both datasets Table 2.2 gives the performance of the current top leaderboard entry (for which an accompanying publication was available) as well for CenterTrack (which does very well on many different datasets) and the SORT baseline.

When looking at Table 2.2, we can first observe that CenterTrack is always either at the very top or very close to it, all while achieving almost real-time performance at ≈ 20 FPS. Only on MOT17 UnsupTrack [KPG20], an extension of stationary image CenterTrack with a re-identification metric trained in unsupervised manner, does at tiny bit better. Although SORT lags significantly behind in terms of scores, one must take into account that it is an extremely simple model that acts exclusively on the bounding boxes of

[4] https://motchallenge.net/results/MOT17/, accessed: 2020-07-21
[5] http://www.cvlibs.net/datasets/kitti/eval_tracking.php, accessed: 2020-07-21

Dataset	Model	MOTA	MOTP	FPS
MOT17	UnsupTrack	61.7	78.3	2
MOT17	CenterTrack	61.4	-	17.5
MOT17	SORT	43.1	77.8	143 - D
KITTI	CenterTrack	88.4	85	22
KITTI	SORT	60.2	72.3	260 - D

Table 2.2: Performance of top models (and SORT baseline) on MOT17 and KITTI benchmarks. Note that the FPS for SORT do not include computing the detections (-D). Further, FPS are not directly comparable due to different hardware.

the (slightly outdated, i.e. generated by models that are not state-of-the-art anymore) public detections provided by the benchmarks. Contrarily, CenterTrack runs what is essentially a fully fledged modern detection CNN. Further, MOT17 is a dataset where re-identification features are particularly important and KITTI has a rather low frame rate; both certainly less than ideal conditions for SORT. However, notice also how SORT itself is extremely cheap to compute, hence easily leading to a real-time tracker when using one of the fast detectors discussed in Section 2.2.9.

The recently proposed Waymo[6] and nuScenes[7] autonomous driving datasets also hosted tracking challenges. On the former, the best entry (with reference) is Quasi-Dense R101[8] with a MOTA of 0.51 for easy objects and 0.45 for harder ones. This very interesting approach computes similarity metrics not only for final detections but quasi-densely for hundreds of region proposals. The best camera-based model for nuScenes is again CenterTrack with an AMOTA@0.2 (a weighted average of the MOTA for different output thresholds; see [ZKK20] for details) of 27.8 for 3D-tracking (as reported by the corresponding paper, no entry on the official leaderboard).

2.4 Relevant Datasets

Training powerful deep learning models requires a lot of data. Hence, we will now take a look at several driving related datasets with significant amounts of night images. Some of those will be extensively utilized throughout this project.

2.4.1 Publicly Available Datasets

Fortunately, using CNNs to solve various autonomous driving tasks is currently a very hot research topic. Therefore many big datasets have already been collected and made publicly available for research purposes. We now discuss the ones most relevant to our particular endeavor (i.e. with much night data).

[6] https://waymo.com/open/challenges/2d-tracking, accessed: 2020-09-04
[7] https://www.nuscenes.org/tracking, accessed: 2020-09-04
[8] https://github.com/sysmm/quasi-dense, accessed: 2020-07-23

First and foremost, there is the Berkeley Deep Drive dataset [YCW+20] or BDD for short. It consists of 100000 independent video sequences recorded in a variety of traffic situations, during different times of the day and with all kinds of weather conditions. Bounding box annotations for 10 classes (including cars, pedestrians, traffic signs and others) are available for a single frame of every video (BDD-det). This makes a very big set of uncorrelated images, which are extremely useful for training detection models. In fact, this is also what sets BDD apart from all other datasets of similar scale, which provide mostly highly correlated annotations for continuous videos. Additionally, BDD also provides such dense annotations for 2000 of the videos, which are about 40 seconds long with 5 FPS. We call this part of the dataset BDD-track. Especially interesting for us is also that almost half of the images (of both BDD and BDD-track) depict scenes in the dark. All together, this makes BDD certainly the most useful dataset for our purpose and hence it forms the basis for numerous training experiments.

Next, we want to mention VIPER [RHK17], a synthetic dataset of driving scenes in a fairly realistic video game. Since the annotations were extracted directly from the game's memory, they are essentially perfect. This is very much unlike real images annotated by humans, which typically contain noticeable inaccuracies, especially on small objects. While real data is usually preferred, experiments have shown that models trained on VIPER generalize reasonably well to actual images. The total dataset comprises 250000 frames, a considerate portion of them at nighttime. Although not very relevant for this project, information such as 3D scene layout or optical flow, which are both very difficult to determine by hand, is also available for all frames.

Finally, there are the Waymo and the nuScenes datasets, which were already mentioned during the discussions of state-of-the-art traffic detection and tracking performance. We mention both of these datasets together as they serve a somewhat similar purpose, that is to develop models which fuse the inputs of different sensors. In contrast to BDD, they contain synchronized recordings of several cameras facing different directions and of multiple LIDAR sensors. Due to the latter, the objects are also annotated with precise depth information via 3D bounding boxes. Waymo has 2000 scenes recorded at 10 Hertz whereas nuScenes contains 1000 annotated at 2 Hertz. While the 20 second long clips are fairly diverse, the day-night distribution is not nearly as balanced as in BDD. Note also that Waymo and nuScenes are tracking datasets (with highly correlated frames) and thus are not a full replacement for BDD-det. However, despite not being their primary goal, they can certainly also be used for training 2D tracking models, especially in combination with BDD-track as all three together form a massive dataset.

2.4.2 CarVisionLight Data

In prior work during the CarVisionLight project there was also a new nighttime driving dataset compiled [GSG20], which we will henceforth refer to as the CVL dataset (or sometimes just as CVL). Refer to Figure 2.10 to see a few example images. Developing a solution that performs well specifically on this dataset is also one of the primary objectives of this thesis.

The CVL dataset consists of several dozens of high resolution scenes recorded at night on Austrian roads, mostly in rural areas. Additionally, several of the videos were taken with stereo cameras. The labeling focuses on moving vehicles, but many parking cars, some traffic signs and pedestrians are also annotated. At first, CVL might seem a little redundant in the light of the various orders of magnitude larger open datasets mentioned in Section 2.4.1, but the type of videos it contains is quite unique. The big datasets consist almost exclusively of city and highway scenes, whereas rural driving situations, like the ones in CVL, are basically nonexistent. Such scenes are generally much darker as there is less peripheral light on the countryside thereby making reliable object detection considerably harder. On the one hand, there is typically also less traffic. On the other hand, cars are often already visible from further away due a bigger and less obstructed (e.g. by buildings) field of view. Adding to that, those far away vehicles are especially important to be detected early to adjust the high-beam light in time and not blind oncoming drivers. This is in stark contrast to city situations where small objects in the distance have rather low importance as there are usually several much more immediate obstacles in form of close cars, pedestrians or traffic signs. Additionally, the driving speed on rural roads is considerably higher, which manifests itself in form of increased motion blur when passing cars driving in the opposite direction.

All in all, the small but high quality CVL dataset poses a unique challenge and it will be interesting to explore how models trained on the big open city datasets can be transferred to work well also in rural areas.

Figure 2.10: Example images of the CVL dataset.

2.5 Other Related Work

So far we have presented discussions on modern object detection and tracking systems in general (see Sections 2.2 and 2.3), but not necessarily with a special focus on nighttime operation. This is because the underlying neural network models are essentially the same and the main difference is the data they are being trained on (see Section 2.4). Still, there are a few other interesting related research directions that we will briefly lay out in this final part of the chapter.

2.5.1 Day-Night Domain Adaption

An intriguing field of neural network research is the science of trying to generalize models to situations they have not been explicitly trained on or in which there is little to none training data available. This type of work is known as *domain adaption*. Particularly interesting for us is of course the adaption of a daytime model to nighttime.

[DVG18] consider a scenario (although for the problem of image segmentation, i.e. assigning a class to every single pixel) where almost only unlabeled night frames are available. Even though the initial model trained on a big daytime dataset performs poorly during late night, its predictions during dawn or dusk are still somewhat reasonable. This is exploited by the authors. They group the night data into varying levels of twilight and then fine-tune their model iteratively on successively darker images in a self-supervised manner. More concretely, they start off by predicting pseudo labels on the least dark twilight images. These are subsequently added to the training dataset followed by fine-tuning of the model. Then the same process is repeated for the other twilight levels in order of increasing darkness. In the end, their model performs significantly better than the pure daytime model despite not having seen any real night labels.

This method is refined and also augmented with generated night data (refer to Section 2.5.2) in [SDG19] to yield even better results. A similar work is also [HDS+19], where the authors generalize from purely synthetic to real videos recorded during thick fog.

2.5.2 Day-Night Image Transfer

In the absence of sufficiently many real (labeled) night images one could also try to generate artificial ones. Specifically, one could take labeled daytime images and attempt to transform them into corresponding images at night, i.e. so that the labels are preserved. Since traffic scenes are not just a bit darker at night but many additional lamps (for example of cars) may completely change the entire lighting in highly non-trivial ways, transforming existing day images into convincing night data seems difficult. However, recent advancement of deep generative models, in particular *generative adversarial networks* [GPAM+14] (or GANs for short), have already shown surprisingly good results for exactly that application.

One popular method is Cycle-GAN [ZPIE17], which is designed to translate images from one domain to another and can be trained on two sets of unpaired (and unlabeled) images.

The key to this method is the introduction of cycle consistency terms to the GAN loss function. Let $x \in X$ be an image from the first domain and $y \in Y$ an image from the second one. Further let $f : X \to Y$ be a mapping (in reality a deep convolutional neural network) from X to Y and $g : Y \to X$ one from Y to X. The *cycle consistency loss* is then given by (2.12). The probabilistic expectations $\mathbb{E}_x$ and $\mathbb{E}_y$ indicate that we wish to optimize $\mathcal{L}_{cyc}$ over the entire distributions of X and Y. In practice, this is approximated by summing over the training examples in a mini-batch.

$$\mathcal{L}_{cyc} = \mathbb{E}_x||x - g(f(x))||_1 + \mathbb{E}_y||y - f(g(y))||_1 \tag{2.12}$$

In words, it guides the model towards finding f and g that can not only transform elements of domain X to domain Y (and vice-versa) but are also consistent with each other, e.g. $f(x)$ can be converted back to x via $g(f(x))$ and the same for the other direction. Further, two separate GAN discriminator networks attempt to distinguish real images from each domain and translated images from the other. Overall, Cycle-GAN leads to remarkable results in a variety of different settings including not only day-night but also seasonal or artistic style transfer. One drawback that must be mentioned is that training requires huge amounts of computation and memory as it involves four different deep CNNs, but three of them can be discarded afterwards and so applying the remaining translation model to entire datasets is definitively feasible.

There also exist several alternative approaches such as for instance [LBK17]. While existing image-to-image translation systems work quite well, at least on lower resolution images, generated images are of course not a full replacement for true target domain data. Figure 2.11 shows some example images. Although they look fairly decent, the resolution is very low. Nevertheless, it is an exciting idea that can certainly be helpful in situations where it is very difficult to gather real data, especially since fine tuning a pretrained translator often already yields decent results with rather few images.

Figure 2.11: Example day-night translations by Cycle-GAN, images are partially rendered at day- and nighttime. `https://junyanz.github.io/CycleGAN/`, accessed: 2020-07-26

CHAPTER 3

Implementation

This chapter covers the major implementation aspects of the thesis project. This includes general considerations such as the choice of suitable data formats, the realization of experiments as well as the development of the final end-to-end tracking tool. Note that the focus of this chapter lies on the actual implementation, whereas detailed results, evaluations and analyses can be found in Chapter 4.

3.1 Development Environment & Choice of Tools

One of the first steps in any project with at least some software component is to choose the right programming languages, frameworks and development environments. A smart choice can make translating ideas into code very swift, while a bad choice can generate a large amount of overhead and significantly slow down the project.

We want to start off by discussing our computer hardware, which is of pivotal importance in a very compute hungry discipline such as machine learning. The majority of development happens on a PC with an Intel i7-7800X CPU with 6 cores and 12 threads clocked at 3.5GHz, 16 GB of DDR4 RAM and an NVIDIA GTX 1080 GPU with 8 GB of VRAM. Essentially all of these specifications are very important for developing state-of-the-art deep neural network models: the CPU and RAM for loading data, the GPU for running CNNs and the amount of VRAM especially also for training. Further, we store big datasets on an SSD allowing orders of magnitude faster memory access, which is again crucial not to bottleneck the training process as well as for efficient general dataset handling. Overall, we have a decent, albeit not top-of-the-line, hardware configuration. Nowadays, much machine learning research happens on large server farms, so having just a single machine is certainly limiting. In particular, we need to be a lot more deliberate with what experiments we run and which models we train for how long. However, we think that exploring what is possible without extraordinary powerful hardware, but just with excellent understanding of state-of-the-art deep learning techniques and smart

planning, is another interesting aspect of this thesis. Finally, we choose Ubuntu 18.04 as our operating system since (as of right now) most machine learning toolboxes are best optimized for Linux.

Next, let us talk about the software stack. The main programming language is Python. Python is a powerful scripting language ideal for quickly writing experimental code with minimal overhead (often caused by mechanisms like class structures or memory management in other languages). It can also be used for developing mature end-user tools. Being interpreted rather than compiled, base Python is not very fast. However, in machine learning applications the execution time is usually dominated by big matrix operations. Those are actually performed by highly optimized libraries such as a basic linear algebra subsystem (short BLAS; on a CPU) or NVIDIA's CUDA (on a GPU) and only invoked via Python interfaces. Hence, we can even use Python for developing our eventual real-time tracking tool for which speed is critical. Further, Python has powerful support for scientific computing, especially also for machine learning, through various mature and widely used open-source libraries. We use matplotlib [Hun07] for creating plots and visualizations, numpy[1] for efficient array operations and Python OpenCV[2] for image processing. For the neural network models, we attempt to stick to the very popular PyTorch framework[3] as much as possible (depending on the available reference implementations) since it is considerably more convenient to use than alternatives such as TensorFlow[4], in particular in the context of research. In general, we try not to reinvent the wheel and adapt existing (and proven) implementations of state-of-the-art models for our purposes, rather than reimplementing them from scratch, whenever possible. Those projects and how exactly they were modified are discussed in the respective sections they become relevant in.

The frameworks mentioned in the previous paragraph usually work smoothly once everything is properly set up, however installing everything correctly, in particular with GPU access, can be quite troublesome. Further, reference implementations of research papers often require different versions of the same dependency, which can also be problematic. We work around these kind of issues by utilizing Docker[5] containers, light-weight virtual machines that can be automatically provisioned with simple setup scripts called Dockerfiles. In fact, many of the big machine learning frameworks already provide official Dockerfiles that automatically set up a container with all complex dependencies already properly configured. Finally, we want to note that most development had to happen in remote fashion due to the global situation while the project was ongoing. While SSH access coupled with a powerful text editor like VIM can already go pretty far, we also heavily utilized Jupyter[6] notebooks (as well as Jupyter Lab). Those are simple development environments that can be accessed over the web browser on a different

[1] `https://numpy.org/`, accessed: 2020-08-02
[2] `https://opencv.org/`, accessed: 2020-08-02
[3] `https://pytorch.org/`, accessed: 2020-08-02
[4] `https://www.tensorflow.org/`, accessed: 2020-09-29
[5] `https://www.docker.com/`, accessed: 2020-08-02
[6] `https://jupyter.org/`, accessed: 2020-08-02

machine. Their interactive nature combined with the ability to display plots, images and videos also makes them excellently suited for data exploration and other experimentation. Thanks to all these great tools, productivity was hardly affected while working from home.

Overall, the choices discussed in this section turned out to be appropriate as we faced almost no unforeseen issues caused by our general toolset.

3.2 Data Handling

This project deals with various datasets that all come in different formats. Additionally, we are interested in developing detection as well as tracking models. Always converting back and forth between inconsistent storage conventions is not just very inconvenient but can also easily lead to hard-to-spot bugs. Further, we want to write generic code that can be reused in other situations rather than working just in one very specific use-case. Clearly, we need to define a fixed data format that is to be used consistently throughout the entire project.

Doing so is, however, not straight forward as some applications are much easier to realize when the data is organized in one format whereas other applications are much easier to implement when the data is available in another format. On the one hand, evaluating an object detector requires the predictions to be grouped by object classes. On the other hand, computing metrics for a tracking algorithm needs the output to be grouped by respective video sequences and further ordered by frame numbers. Since the preferences are already mutually exclusive for just these two applications, doing at least a few conversions will be unavoidable. Typically, the most space efficient way to store data is in form of a hierarchy that aggregates entries with fields of equal value. For example, an organization like (3.1) would allow us to completely avoid saving any redundant values.

$$video \rightarrow frame \rightarrow object\,class \rightarrow prediction \tag{3.1}$$

However, reshaping this hierarchy into, for example, $object\,class \rightarrow video$ or filtering out all predictions of a certain class would be rather inconvenient. In contrast, a completely flat format, i.e. storing everything as a list of $(video, frame, object\,class, prediction)$ tuples, makes all kinds of filtering and grouping operations very simple but comes at the cost of increased redundancy. As we generally do not have to manage exorbitant amounts of predictions but definitively have to perform numerous format conversions (especially during the experimentation phase of the project), we opt for a completely flat representation of labels and predictions.

Having decided the general data layout style, the next question is what information to store for every entity. Obviously, we need the predicted bounding box `bbox`, the object type `category` and a confidence score `score`. In case of labels, the latter is simply

set to 1. For tracking purposes, every prediction must also have an identifier `id` (which is -1 for pure detections). Predictions are related back to the input images via their filename `name` and, if the image is part of a video, their frame number `timestamp`. See listing 3.1 for a full example. In Python, a set of predictions is represented by a list of dictionaries, which can easily be written to disk in form of a JSON-file.

Listing 3.1: Example prediction in our flat data format.

```
{
    name: 'video1',
    timestamp: 118,
    category: 'car',
    bbox: [109, 231, 154, 317], # [x1, y1, x2, y2]
    score: 0.793
}
```

The labels of all datasets used throughout this project were initially converted to this flat format. To maintain matching category names, we stuck to the 10 classes of the BDD-det dataset (plus the additional *ignore* class described in Section 3.3). Several convenience functions for filtering and grouping flat datasets into various hierarchies were also implemented. Altogether, our custom data format proved to be an effective solution that greatly reduced the time spent handling incompatible data sources.

3.2.1 Dataset Filtering

As already mentioned before, one of the core problems of this work is the considerate mismatch between the training and testing data distributions as driving in cities is quite different from driving on rural roads. Hence, one of our first ideas was to bring the training- a bit more in line with the testing-data through intelligent filtering. Although this ultimately did not turn out to be particularly helpful for us, it still involved some interesting approaches that could be useful in other situations. Hence, here follows a short list of some of the ideas that were tried:

1. Not use full tracking data for training detection models but only every i^{th} frame where i is as large as possible since consecutive frames are usually far too similar to meaningfully contribute to the learning process.

2. Filter out boxes with high overlap to other boxes. The idea behind this is to remove the far ends of long lanes of parking cars that are very common in the city but basically nonexistent on the countryside.

3. Remove objects so small that one cannot expect a model to reliably recognize them. We hoped this would reduce the large number of very small false detections that could be observed in our initial experiments.

4. Add “interesting” image crops in full resolution (the full image is typically downscaled for training) to the dataset. In particular, we wanted to put a focus on the regions containing small far away cars (which are particularly important in our domain of interest). More precisely, we required every such image to contain at least one small car without any overlap to another one. This turned out to be a pretty good heuristic for selecting only sub-images containing at least one small driving car (bounding boxes of small parking cars typically overlap a lot).

5. Find driving cars with active lamps via thresholding the average lightness value (in the HSL color space) of all pixels in a bounding box. Alternatively, one could check for a certain fraction of pixels with high lightness if the lamps are expected to be small compared to the total object size.

At least judging by looking at a few examples (see also Figure 3.1), the techniques listed above seemed to “clean up” the training dataset reasonably well, eventually reducing the number of labels by over 50%. Nevertheless, training with the filtered annotations only showed improvements during the first few epochs. Later experiments (see Chapter 4) also revealed that modern neural network models have enough capacity to fit many different kinds of cars and dataset filtering is thus unlikely to help for this particular application. Similar practices could, however, certainly be useful in other scenarios.

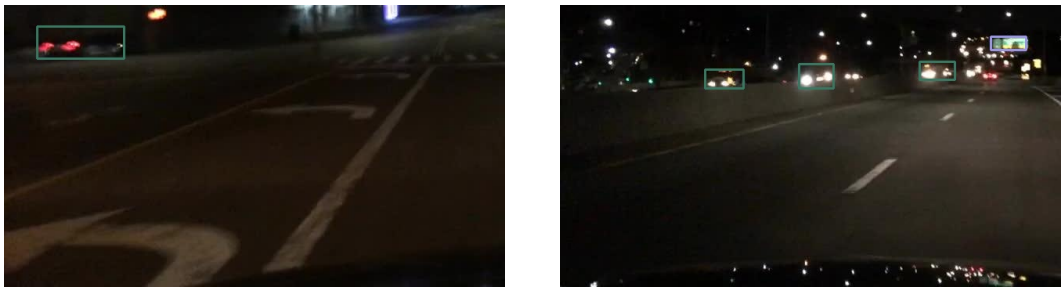


Figure 3.1: Two sample images resulting from filtering BDD-det. Note that both images are full resolution sub-regions as described above.

3.3 Evaluation Framework

Before meaningful experiments can be performed, all evaluation metrics have to be defined and implemented. Since we want to ensure comparability with results reported in literature, we base our evaluation framework as much as possible on established open-source code. All interfaces are based on the data format described in Section 2.4, for ease of use and to ensure consistency between all results obtained throughout this project. In addition to how the evaluation was implemented, the next two subsections will also discuss why the chosen metrics are relevant for this thesis.

One additional technique we apply is the notion of so called *ignore-regions*. Those mark certain parts of an image we do not care about. Declaring image regions that cannot (e.g., because they are too dark or too blurry) or should not (e.g. a parking lot with dozens of cars) be fully annotated helps avoiding bias in the evaluation results (when the model predicts a car that is not annotated or misses an object one cannot expect to be detected). Integrating this feature into the evaluation process is rather straight-forward as it amounts to simply filtering out all predictions that significantly (we choose the threshold 0.5) overlap some ignore region (plus then deleting all of the latter) before computing any metrics. It just needs to be noted that we need to consider the actual overlap, i.e. $|B \cap I|/|B|$, instead of the IoU since ignore-regions I are generally not single object instances, but are usually rather large and can contain many potential boxes B.

3.3.1 Object Detection

For evaluation of object detection models we will use the widely accepted mAP metric (see Section 2.2.2) with an IoU-threshold of 0.5. In our application, it is primarily important that objects are actually detected, the accuracy of the predicted bounding boxes is only secondary. In fact, in many cases, especially for very dark or very far-away objects, not even a human annotator can draw the bounding box with perfect accuracy. Thus, 0.5 overlap is certainly sufficient for nighttime traffic detection. Consequently, we will also not consider the mAP@0.5:0.95 or similar aggregate scores. At this point it also needs to be noted that for the CarVisionLight dataset we will primarily look at the AP for just the car class, as AP values for other object types may not be sufficiently representative due to their low number of occurrences. In addition to the numeric evaluation, we also study pr-curve plots to understand the precision-recall trade-off and potentially reveal any unexpected model behavior.

Our actual implementation is strongly based on the official evaluation code for the BDD dataset[7] (which in turn is based on the official evaluation code of Faster R-CNN[8]) augmented with some adjustments to cope with our data format and the option to also draw pr-curves. As we heavily utilize BDD for training models and because it is one of the best known driving related datasets, we think it is sensible to rely on their particular flavor of mAP calculation.

3.3.2 Multiple Object Tracking

We assess the performance of tracking algorithms using the most relevant subset of the CLEAR-MOT metrics (see Section 2.3.1). Concretely, we primarily consider the precision, the recall, the number of ID-switches and the summarizing MOTA. The first two give an idea about the fraction of missed objects as well as the rate of incorrect detections. The ID-switch count quantifies the consistency of the model in preserving object identities across frames and the MOTA summarizes all of it with a single number. While there

[7] `https://github.com/ucbdrive/bdd100k`, accessed: 2020-08-04
[8] `https://github.com/rbgirshick/py-faster-rcnn`, accessed: 2020-08-04

exist many other performance measures for tracking algorithms, we prefer to focus only on those that correlate well with real world performance in our particular domain of interest (for example, the MOTP is of rather low importance for reasons already outlined in the previous section on detection). Metrics are certainly very informative in general, but they rarely tell the whole story, especially for applications as complex as multiple object tracking. Thus, we also regularly judge our results qualitatively by applying visual inspection, i.e. by carefully studying the output videos. This helps to identify common problems and failure cases that are not directly reflected in the numeric measures.

We compute all metrics using the py-motmetrics library[9], an excellent MOT evaluation framework that carefully handles complicated constellations that can arise when trying to match detections to ground truth objects. More concretely, we first build a `MOTAccumulator` based on the IoU-distance for every video and then let py-motmetrics compute all performance scores from the accumulator's event histories. Finally, we want to mention that some care has to be taken to make sure that we do not forget any tracker updates (for empty detection sets) when dealing with our flat data format that does not inherently store information for frames with no detections.

3.4 Neural Networks

A significant part of this work is training and testing various deep learning models. However, the main focus here lies on trying to adapt state-of-the-art approaches to our novel domain (primarily via variation of training data and strategies) rather than coming up with new network architectures or extensively tuning the numerous hyper-parameters connected to a CNN model. To make this process as efficient as possible, we try to utilize existing open-source implementations (with necessary adjustments) wherever we can.

This sections covers the particular models that are studied and why those were chosen as well as which open-source implementation we rely on and how they were configured/-modified.

3.4.1 YOLOv3

The literature study in Section 2.2 makes it clear that solid real-time (a key objective of our work) objection detection, especially on moderate hardware, requires a one-stage detector. However, that does not narrow down our choices very much as a plethora of different detectors is available nowadays and new ones are being published on a regular basis. Although several of them differ substantially in terms of structure, most perform roughly in the same ballpark, i.e. a few percent, on the standard benchmarks. First and foremost, we want to deliver a workable solution for our particular problem, so doing 1 or 2 percent better or worse is not one of our main concerns. Further, there is no guarantee that such minor differences on a general object detection task (like MS-COCO) would actually transfer to traffic data anyways. Therefore, any model scoring close to the

[9] `https://github.com/cheind/py-motmetrics`, accessed: 2020-08-05

current state of the art will probably be sufficient for our experiments. This also means that criteria such as the quality and ease of use of available open-source implementations (as well as pretrained network weights) have a much bigger influence on our selection.

After careful consideration, our first choice falls on the very popular object detector YOLOv3 [RF18]. This model is not designed to squeeze out every last percent of mAP on standard benchmarks or to minimize highly theoretical throughput numbers, but to provide a good trade-off between speed and accuracy on real-world problems, in particular also when retraining on custom datasets. This fully matches our goals. Additionally, there are several excellent open-source implementations of YOLOv3. We worked with the PyTorch-YOLOv3 repository by Erik Linder-Norén [10]. We preferred this version over the official one by the YOLOv3 authors [11] due to it being written in PyTorch, rather than in a custom low-level C (the programming language) framework, and thus much easier to run and especially to modify. Lastly, at this point it needs to be mentioned that YOLOv3 was indeed one of the best object detection models at the start of this project and was only superseded by EfficientDet and YOLOv4/v5 very recently. Although our results for YOLOv5 clearly exceed those of YOLOv3, we still want to cover the latter at least briefly as it was the core of many initial experiments which helped to guide the direction of our work.

PyTorch-YOLOv3 is already quite well suited for our use-case and there were only a few minor things that had to be adjusted. First, we added a way of outputting predictions not just as boxes drawn into the images but also in our JSON format to allow subsequent computational handling of the detections. Further, we extended the training log with class-wise performance scores to supervise not just overall summary metrics but also how the model is improving on the object types we are particularly interested in (e.g. cars) as it is learning. To keep all training experiments consistent, we always train with a nominal batch-size of 16 (which is a sensible choice for this type of model). Though, depending on the size of the images, we cannot always compute the gradients for a full batch at once with just 8 GB of GPU memory. Thus, we must not forget to always increase the gradient accumulation parameter `--gradient_accumulations` by a factor of s as we scale down the batch-size `batch_size` by s. Note that $s = 2^i$ is always a power of 2 to ensure optimal execution speed.

Rectangular Inference

PyTorch-YOLOv3 with the small changes discussed above worked very well for experimentation. However, once it came to developing the first prototype for an actual real-time tracker, we ran into problems. Mainly, accurate detections (especially for smaller cars) required a high image resolution (like 1600×1600) but inference on such large tensors was rather slow. One of the main culprits for this was that the model only worked for square images and thus automatically "letterboxed" (i.e. added big gray stripes at the

[10] `https://github.com/eriklindernoren/PyTorch-YOLOv3`, accessed: 2020-08-08
[11] `https://github.com/AlexeyAB/darknet`, accessed: 2020-08-08

top and the bottom) our almost $2:1$ ratio frames into squares. This meant almost 50% of the neural network computation was completely wasted on purely gray pixels. Fortunately, there is nothing stopping the general YOLOv3 architecture from predicting on an $S \times T$ rather than an $S \times S$ grid (as long as some minor size constraints between S and T for upsampling layers are fulfilled). Even more, it is in the nature of CNNs that the same model can be applied to almost any (again with some architectural restrictions) image shape. This means we can keep training on square images (which is much easier when the data does not have uniform aspect ratios) but almost double the inference speed by computing predictions on rectangular images without any loss in detection quality. See figure 3.2 for a comparison between a CVL image padded to a square and a rectangle compatible with the YOLOv3 model. There is a huge difference in the number of completely uninformative gray pixels.

Figure 3.2: A CVL image padded to a square and to a $2:1$ rectangle.

In theory, extending a YOLOv3 implementation to properly handle rectangular images should be rather straight-forward. In practice, it, however, turned out to be considerably more difficult than originally expected. A lot of tensor shapes all throughout the model had to be meticulously adjusted, which was further complicated by slightly confusing naming schemes in the existing source code. A particularly tricky part was properly padding up images to fit into an $S \times T$ grid as well as eventually undoing the padding to return bounding boxes with respect to the original image. Let us assume that we want to fit a $w \times h$ image into a $w' \times h'$ shape that is compatible with an $S \times T$ output grid. First, we have to compute $s_w = w'/w$ and $s_h = h'/h$. Then the joint scaling factor is given by $s = \min\{s_w, s_h\}$. At last, we scale the image by s in both dimensions and pad it up to size $w' \times h'$. For simplicity, we apply one-sided padding to the right and bottom, which also has the additional advantage that it can be ignored when transforming detection coordinates back to the input image. In the end, rectangular inference worked very smoothly and indeed sped up detection time by almost a factor of 2.

3.4.2 CenterTrack

Although our primary focus is set on the tracking by detection framework for reasons outlined in Section 3.5, we still want to test at least one of the recently published pure neural network trackers. Looking at the current leaderboards of relevant tracking benchmarks (see Section 2.3.1) reveals CenterTrack [ZKK20] as a particularly promising model that seems to perform remarkably well almost everywhere. The authors also provide an excellent open-source implementation plus various pretrained network weights for different tasks [12]. In particular, there exists one model trained on the driving dataset nuScenes [CBL+19], which also contains night data (although less than BDD). This is particularly convenient since retraining/finetuning CenterTrack is very resource intensive (requiring large batch-sizes, multiple GPUs and long training times) and likely would not have worked very well on our development hardware. Hence, we would first try out this pretrained model and only worry about further training with other data if the initial results seemed particularly promising (in general, one would not expect dramatic differences between models trained on BDD and on nuScenes).

Getting the authors' code to run properly was a bit of a challenge due some version mismatches in our installed libraries, however this could be addressed quite swiftly by setting up a new Docker container. We also had to implement our own script for benchmarking CenterTrack on the CVL dataset and persisting all results. While the nuScenes instance of CenterTrack is actually designed for 3D-tracking, it also automatically returns appropriately projected 2D-boxes, which are then considered by our standard 2D evaluation. Note also that we reduce the tracking threshold from 0.3 to 0.1 to improve CenterTrack's recall, one of our most important metrics.

3.4.3 YOLOv5

As the work on this thesis was progressing, several new object detection methods claiming significant performance improvements were published. Obviously, those had to be tried too. Since we already committed to working with YOLOv3, it was natural to also look at the successors YOLOv4 [BWL20] and YOLOv5 [13]. The reason why two successor models were released in such a short period of time is that they were developed by different authors. In fact, the creators of YOLOv5 have no relation to those of YOLOv3. This combined with some apparently not 100% fair performance comparisons has lead to some controversy around YOLOv5. However, as already discussed in Section 3.4.1, we are primarily concerned with real-world performance when retraining on a different dataset as well as ease of use and modifiability of the public implementation. YOLOv5 is excellent in all of those aspects, which is why we opted to use it over YOLOv4 (even though the latter does a little bit better on MS-COCO).

To achieve optimal performance, several decisions/adjustments had to be made. First, we decided to base our training experiments on the M-version of YOLOv5. It still easily ran

[12] https://github.com/xingyizhou/CenterTrack, accessed: 2020-08-09
[13] https://github.com/ultralytics/yolov5, accessed: 2020-08-09

in real-time while achieving a significantly higher mAP on MS-COCO than the smallest (and fastest) S-version. Exchanging another $\approx 33\%$ of speed for an only modest increase in prediction quality by using the L-version did not seem worth it (nevertheless, we also experimented with this version for a bit). We did not recalculate any anchors when retraining the preexisting models on new datasets (option `--noautoanchor`) as we found this to dramatically increase convergence time. The general purpose COCO-anchors seemed to transfer very well to traffic data as we observed the validation mAP rising sharply already in the first few training epochs. Next, we switched off mosaic augmentation (option `--rect`). While this is a very interesting new idea, it is unfortunately actually counter-productive when training with small objects, like there are hundreds of thousands of in BDD, since this technique scales down images by another factor of 2 (see also Section 2.2.8) thereby making already hard to spot objects even more difficult to recognize. One final issue that deserves to be mentioned is an important fix suggested by the YOLOv5 authors for continuing training more seamlessly. YOLOv5 utilizes a technique known as *exponential model average* (EMA), i.e. using an exponentially decaying average of all individual models (after every weight update) for computing predictions (rather than only the most current weights). This helps smoothing out performance oscillations as the model learns, especially when it is close to converging, but unfortunately makes restarting canceled training jobs a bit troublesome (with the original YOLOv5 implementation at least). However, after several bug-fixes, most crucially not always resetting the number of EMA updates to prevent renewed burn-in, leading to amplified instabilities, this could be resolved for the most part.

In-depth experiments (see Chapter 4) revealed YOLOv5 as a very powerful model that is well suited for our task of accurate but real-time object detection. This is also why it became the backbone of our eventual tracking tool. More details on the training process for the final model are given in the corresponding Section 3.6.2.

3.5 Tracker

Once we had trained the first versions of our nighttime detection models, the next step was extending them to full tracking systems that are able to preserve object identities. A straight-forward approach to do so is the tracking by detection framework discussed in Section 2.3.

In particular, we first tried the most straight-forward method of this type that is SORT [BGO+16]. Although this rather simple approach which operates exclusively on the bounding boxes returned by the detection model is nowadays far from state-of-the-art on challenging traffic benchmarks, we considered it well suited for our particular domain. To start off, we are working in a high frame rate regime (30 FPS) where objects typically only move very little in between frames and association of closely overlapping bounding boxes should work quite reliably. Additionally, countryside road data is for the most part rather sparsely populated with objects, advanced schemes primarily aimed at reidentifying strongly overlapping and disappearing objects in chaotic crowds are

most likely not required. Worse, it might even slow down the tracker through additional network evaluations or complex matching algorithms. Contrary, SORT is very fast (especially if typical scenes contain only a handful of object candidates) and causes very little additional overhead. So all in all, SORT appeared, despite its simplicity, well suited for our task at hand. This is confirmed by a simple experiment where a basic SORT tracker applied to the ground truth object labels (i.e. the boxes but without IDs) reaches well above 95% MOTA on the CVL dataset. The majority of mistakes are due to the fact that SORT only starts tracking (i.e. outputting IDs) objects once they have been detected in several consecutive frames.

Initially, these results might suggest that (for our task) the tracking performance is entirely dependent on the detection quality. While there is certainly a very strong correlation, a good tracking component can certainly make up for some detection errors. As a simple example, displaying tracks only after consistently observing the same object for a fixed amount of time can significantly help with reducing spurious detection errors, which could be very dangerous in the context of autonomous vehicles. This is the primary (and also really the only) error correction technique employed by the SORT baseline [14], but other improvements can be made in that regard. In particular, the next section will propose several extensions to the basic SORT method that turned out effective in the domain considered by this thesis. They were developed after studying the concrete tracking behavior, not just through metrics but also via careful visual inspection. Overall, they noticeably improved the tracking performance on the CVL evaluation data, especially qualitatively.

3.5.1 SORT$^+$ (Extensions to SORT)

This section gives a detailed description of various changes that were made to the standard SORT algorithm. We will use the name SORT$^+$ to refer to this improved version.

Spawn Confidence

Perhaps the most frequent issue observed on CVL data is that correct tracks suddenly disappear as the prediction confidence for the corresponding object drops below some threshold. This is most likely due to nighttime prediction models not being as reliable and smooth as their daytime counterparts. Hence, rather small image changes can result in sharp drops of prediction confidence, or in the worst case even complete failure to detect an object. Standard SORT allows mitigating this problem by adjusting the global threshold τ that is used to filter away uncertain detections. However, while lower τ usually means fewer lost tracks, it at the same time also tends to result in more false ones.

To overcome this dilemma we introduce an additional confidence threshold τ_s that controls when new tracks are spawned. In other words, new track candidates (that are in turn

[14] `https://github.com/abewley/sort`, accessed: 2020-08-11

only displayed after several successful hits) are only created for detections with confidence greater than τ_s but existing tracks (including ones not yet shown) are extended with all boxes of confidence greater than τ. With appropriately chosen threshold values this noticeably improves tracking consistency without introducing too many additional false positives.

Displaying Kalman Propagation

SORT models existing tracks with simple linear motion through Kalman filters. At every timestep the most probable bounding box for each track is estimated according to the current state of the corresponding Kalman filter. New neural network detections are then matched to those estimates. Unmatched tracks are kept alive for a certain number of frames before being deleted, but remain invisible when there are no network detections. This choice was made to handle simple cases of reappearance after occlusion (common benchmarks generally only have annotations for visible objects) [BGO+16].

In city environments it is quite common for objects to disappear temporarily, but in the rural CVL data such situations are rather rare. Further, knowing the approximate location of currently invisible objects could be very useful in the context of intelligent vehicles. As already discussed previously, the tracker seems to have problems with continuously maintaining tracks over longer periods of time due to random confidence drops in the detector. Taking everything mentioned so far into account, displaying propagated Kalman states with no matched detections for at least a few (we denote this number by n_f) frames appears to be a good idea. In our setting, it helps smoothing out trajectories while only introducing insignificant amounts of new errors. To avoid extending completely false tracks for too long, one can also enable the Kalman state display only after a track has been alive for a certain amount of frames or if the most recent matched detection exceeded some specific confidence level. We did, however, not observe significant improvements with the extra strategies mentioned in the previous sentence and thus stuck to the simple version of the Kalman output feature discussed before.

High Confidence Detections

One of the most important aspects of our nighttime vision system is real-time performance to ensure fast reaction times of an automatic control/assistance system built on top of the fundamental tracking algorithm. With that goal in mind, showing objects only after detection in several consecutive frames, and thereby introducing extra latency, may seem counter-productive. However, it is necessary to avoid picking up too many false tracks when operating in the dark, where it is easy to misinterpret light reflections or unusual black outlines as objects, even for humans.

We noticed that our CNN model achieved high precision for bigger confidence thresholds (e.g. 0.75), i.e. almost all detections were correct. Thus waiting to collect evidence over multiple frames for objects detected with such high confidence only accumulates

additional delay without really improving the accuracy. This is why we introduced another parameter τ_h to our tracker, where detections with confidence larger than τ_h are displayed immediately. We want to note that closer and bigger objects tend to be detected more confidently than smaller and further away ones. Coincidentally, shorter reaction times are a lot more important for closer objects whereas not constantly outputting false (non-existent) boxes is the priority for further away ones. The τ_h-feature further aligns the tracker with this desired behavior.

Overlap Pruning

Finally, one common problem we observed when testing various different nighttime detection models was that they would sometimes detect multiple boxes around the same car, which were however different enough to not get dropped by NMS. One prominent example is when a big close-up car is detected but the model also believes that both of its head lights individually are (small) vehicles.

Fortunately, these cases can be filtered out quite reliably with a strategy that we call *overlap pruning*. In a given frame we delete all predicted boxes B for which there exists another box B' such that the relative overlap of B with B', i.e. $|B \cap B'| \,/\, |B|$ is greater than some threshold t_o. The deletion is performed in order of increasing confidence to avoid situations where some object o is not dropped because the object o' containing it had already been deleted in a prior iteration. The procedure works reasonably well as bigger objects tend to have higher confidence values than smaller ones and overlap pruning thus behaves as expected. Note that in our type of data the number of situations where two correct objects have high overlap is much smaller than the amount of incorrect highly overlapping predictions (discussed in the previous paragraph). Hence, this strategy helps to improve the performance. One could also exempt high confidence predictions from the pruning process to make it less likely that relevant predictions are accidentally dropped in domains where such cases are more frequent.

Figure 3.3: Left: An example of incorrect detector behavior observed frequently with not well trained models. Right: Sanitized output after overlap pruning.

3.6 End-to-end Tracking Tool

Finally, we put our insights together to build a prototype for an *end-to-end* tracking tool. In this context, end-to-end means that the program directly processes videos and outputs final tracking information. There are no intermediate steps of moving around temporary results and running several scrips as it was common during the research phase of this project. The tool represents a prototype for a nighttime tracker that in the future may be extended or integrated into other systems operating on the tracking results, for example an intelligent lighting controller. Further, a meaningful speed test to assess the tracker's real-time capabilities also requires a full implementation of the entire pipeline.

The subsequent sections document the structure and realization of this final tracking tool. The results of various experiments which formed the basis for key decisions taken here can be found later in Chapter 4.

3.6.1 Input & Output

When developing software one of the key design decision is often its interface, i.e. how it communicates with the outside world. What are the inputs, what are the outputs and how can a user interact with the program?

Since our tool is primarily intended to be a prototype for performance evaluation as well as a reference for further extension and reuse in advanced systems building upon the tracking information, rather than to be used as it is by end-users, we consider a simple command-line interface to be sufficient. The full calling syntax is given in Listing 3.6.1 below. We want this to be a self-contained tool that can be applied as it is rather than a research script that requires the user to specify numerous hyper-parameters. Hence, we kept the interface as concise as possible (all parameters can be reconfigured by editing corresponding constants in the source code).

```
python track.py --input INPUT --output OUTPUT [--real]
```

The tool should output tracking results both visually in form of the original videos with drawn bounding boxes as well as numerically in a format that programs (for example an evaluation script) can continue working with. For the latter, we create JSON-files in our custom project format (see Section 3.2). However, this type of data organization can be a bit troublesome to handle for applications that were not initially designed to be conform with it, primarily due to the omission of frames without detections. Hence, we additionally save the tracking results in a nested format, i.e. a list with one entry for every frame that is another list of detections (with our standard attributes). Individual lists can be empty if no objects were recognized in the corresponding video frame. For the visual output, we indicate the object ID by the color of the bounding box, i.e. the same object has always the same box color, and we also output the object's class and its predicted confidence. An example output of the tracking tool can be seen in Figure 3.4.

To avoid having to always convert back and forth videos when wanting to test the tracker with new data, our tools needs to be flexible with respect to the type of its inputs. More

Figure 3.4: Example output of the tracking program for two frames of the same video (cropped for better visibility). The same object instances have the same box colors. The character 'c' indicates that both objects are correctly recognized as cars.

concretely, our tool can work with folders containing numbered images of video frames (in common image formats) as well as video files, again in established encodings. Those are specified with the `--input` option. Fortunately, this is rather simple to realize using OpenCV's excellent `cv2.imread()` as well as `cv2.VideoCapture()` functionalities. It also needs to be mentioned that the tracker is seamlessly able to handle videos of different resolutions as it automatically performs appropriate down-scaling with respect to the neural network model. This aspect is discussed further in Section 3.6.2.

Camera Simulation

While our tool is designed for use with existing files, in reality a similar system could be applied to a live camera feed. This is not a goal of our current project, but we still check how our software would perform in such a setup in Section 4.3.3. Note that a system processing on average about 30 frames per second is not necessarily equivalent to one that performs in real-time on a continuous image stream. Concretely, there could be

a significant variation in processing times for different frames, e.g. one frame f_l takes $9/30^{\text{th}}$ of a second to process whereas 9 other frames take only $0.111/30^{\text{th}}$ (and $1/30^{\text{th}}$ for the remaining 20). When running such an algorithm on real camera input, receiving f_l would result in the program having to skip the next 8 frames. Additionally, there is nothing to be gained from the faster frames as the tracker always has to wait for the next image to become available anyways. We simulate this behavior (tool option `--real`) by first loading all frames of a video into memory and then supplying the next frame only every $1/30^{\text{th}}$ fraction of a second while skipping ones that have not been polled in time. Although this strategy is certainly very memory intensive (this could, of course, be drastically improved with asynchronous loading) and should only be used for short videos, it is sufficient for testing the tracker's real-world behavior. For normal testing purposes (i.e. when not explicitly wanting to simulate a camera feed) one should use the standard input scheme where images are fetched as needed one after the other.

3.6.2 The System

Having described the input and output, we can now move on to a discussion of the core tracking system. On a high level, it consists of three main steps: first preprocessing (cropping and resizing) the input frame, then running it through a CNN to compute object detections and at last using those to update a tracking algorithm that returns the final output boxes. The subsequent sections cover all three components, and a visualization of the system's full workflow is given by Figure 3.5.

Preprocessing

Before we can apply the tracker's main CNN to a given frame we need to first transform the image slightly, primarily in terms of resizing it to the optimal resolution. This is crucial to achieve real-time performance since albeit the used CNN is very fast, it is not quite efficient enough to run directly on full-HD 1920×1080 videos. Some experimentation showed that the tracker can comfortably reach ≈ 35 FPS (including not just the CNN pass but also any other preparation and tracking computations) with a resolution of 1280×640 on our hardware. Further, we did not observe a major drop in detection performance when going down to this resolution, which is probably due to the fact that this is very close to the maximum resolution of BDD, our training data. However, naively resizing input images to this resolution is only a good idea if they are already in approximately a $2:1$ aspect ratio (which is not the case for all of the CVL data). Otherwise, we would waste a lot of pixels through letterboxing as already described in Section 3.4.1. Instead, we try to determine the optimal image shape with roughly $P = 1280 \cdot 640$ pixels such that any border padding is minimal. It must also be taken into account that both the width and height of the image have to be multiples of 32 for the YOLOv5 architecture to work.

More precisely, we want to resize a $w \times h$ image uniformly by a scaling factor s such that $w' = w \cdot s$, $h' = h \cdot s$ and $w' \cdot h' = P$. Plugging in these constraints into a single equation yields $w \cdot h \cdot s^2 = P$ with the solution for s given by Equation 3.2, from which

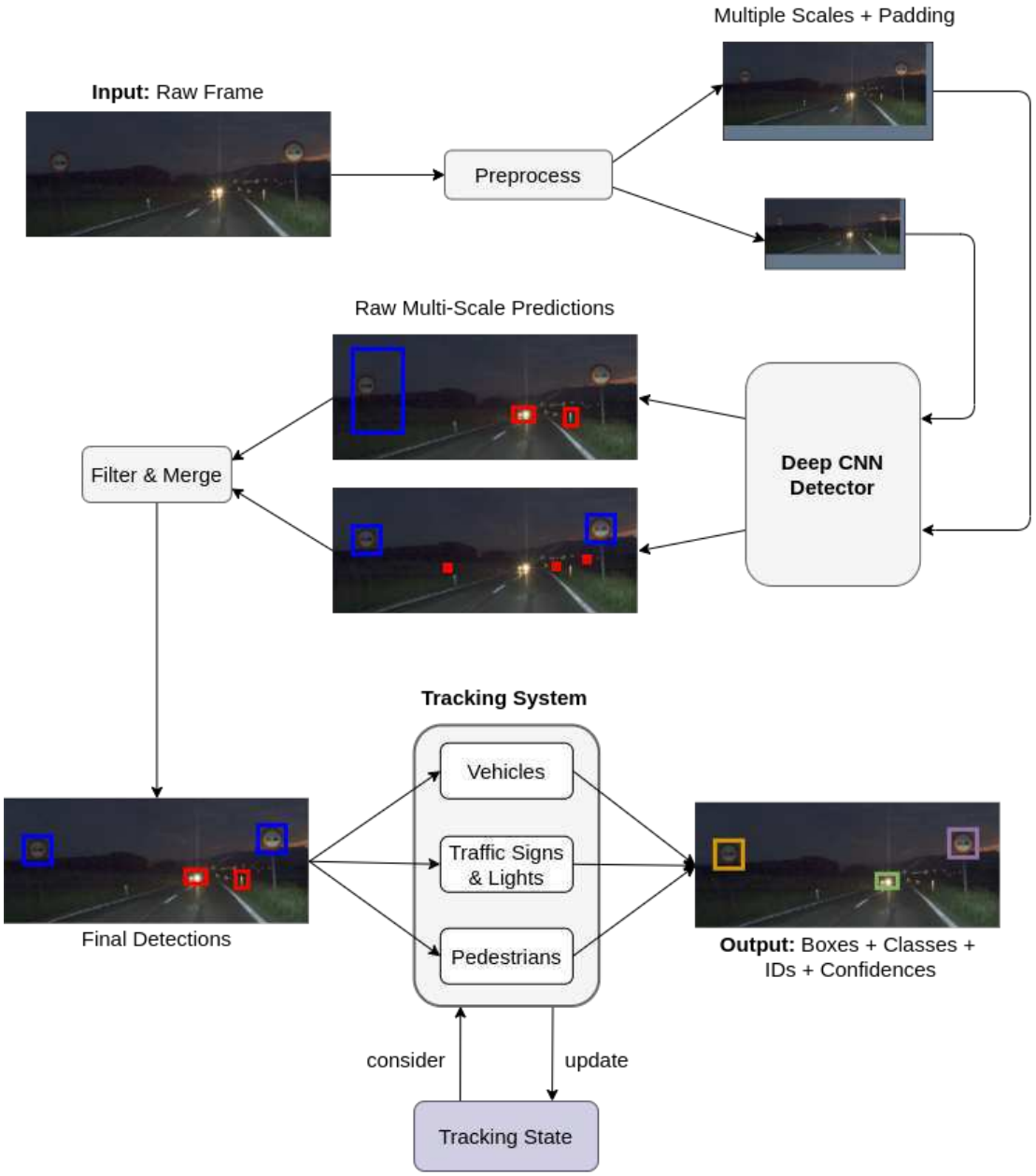


Figure 3.5: The full workflow of the tracking system.

then w' and h' can be easily determined. The final size compatible with the YOLO grid is then computed by $x'' = 32 \cdot \lceil x'/32 \rceil$ for $x' = w'$ and $x' = h'$. While the resulting image may ultimately have slightly more than P pixels, this is not an issue as it hardly affects performance (alternatively we could just reduce the size by 32 in each dimension).

$$s = \sqrt{\frac{P}{wh}} \tag{3.2}$$

We also implemented a way for the model to perform predictions on multiple different resolutions and then eventually merge them together via NMS. The underlying idea is that the model might be more accurate on big objects when the resolution is lower. Further, small objects most of the time only occur in a rather limited field of view (see also Section 3.6.3). This makes it possible to execute two separate neural network runs for a smaller high resolution 1280×448 sub-region as well as for a 640×320 version of the whole frame while still staying essentially real-time. When combining the results of both runs one needs to first filter out big objects from the high resolution predictions and small ones from the low resolution detections as those are less reliable for the respective network inputs. While we found that merging predictions at different scales was helpful for some of our earlier (and weaker) models, we did not observe any improvements for our final YOLOv5 model, which is detailed in the next section. Hence, we ultimately disabled the multi-resolution technique.

Finally, after all resizing is done, the `uint8` (unsigned 8-bit integer) arrays must be transformed to floating point data with values in $[0, 1]$ through a datatype conversion followed by a division with 255.

Neural Network Model

The second step of the tracking pipeline is to apply a deep CNN detection model on the properly preprocessed image leading to object detections, i.e. bounding boxes and confidence values. This is perhaps the most critical step of the entire tracking tool as the detections need to be accurate but at the same time also quick enough to compute. The neural network computations make up for more than 90% of the frame-by-frame run-time. After our training experiments (documented in Chapter 4) we came up with the neural network model described next.

At its core our neural network is a YOLOv5-M model[15]. The M variant was selected as it showed the best trade-off between inference time and detection performance. Starting off with the weights, pretrained on MS-COCO, provided by the authors, we trained the model for 100 epochs on the full BDD detection dataset at 640×640 resolution using the default hyper-parameters (for some more training tweaks that were applied see Section 3.4.3). Then we dropped the initial learning rate by a factor of 100 and finetuned for 25 more epochs, again on BDD but this time with the full 1280×1280 resolution. We initially train with a lower resolution because it is more than a factor of 4 faster, more stable and the model is still learning very well. Nevertheless, the subsequent finetuning on higher resolution leads to significant further gains. Though, it appears to converge rather quickly, hence we stop training after just 25 epochs. Note also that the decreasing

[15] `https://github.com/ultralytics/yolov5`, accessed: 2020-10-06

of the learning rate is critical to avoid a significant loss of performance in the early epochs of finetuning, which takes a long time to recover from. Since we train on the full BDD dataset, i.e. also on many daytime images, we found that the network works just as well during the day as it does during the night. This is not our specific aim, but it is certainly an additional advantage that the nighttime tracking system does not break down when the sun is shining.

Overall, we found that the CNN yields very good detection results while maintaining fast inference times (see Section 4.3), together making it a strong backbone for our tracking system. Should there be any significant object detection advances in the coming years, this component could also be replaced easily.

Tracking

The final part of our tool is where the actual tracking happens. The boxes predicted by the CNN in the previous step are scaled back to the original input image size (the network runs on a downscaled version as described in Section 3.6.2) before being passed on to the actual tracking component which temporally associates detections in between frames.

For that purpose we utilize SORT$^+$, i.e. SORT with our domain specific extensions described in Section 3.5. To be more precise, we actually utilize three tracker instances for different subsets of object types. This is done because the neural network does not behave equally for all types of objects. Thus, to achieve the best possible performance we want to use distinct SORT$^+$ configurations for each tracking level. A full enumeration of all parameter settings is provided in Table 3.1.

Object types	τ	$a_{\max}$	$h_{\min}$	τ_s	τ_h	n_f
Cars, trucks, bikes, motorbikes	.25	10	3	.50	.75	2
Traffic signs, traffic lights	.25	5	5	.50	.90	1
Persons	.25	5	2	.25	.50	1

Table 3.1: SORT$^+$ parameters used for tracking objects of different types. $a_{\max}$ denotes the number of frames without detection after which a track is deleted, $h_{\min}$ is the number of hits necessary before a track is shown and the other parameters are as defined in Section 3.5.

In general, the detection of vehicles is by far the most reliable, hence we choose a lower number of minimum hits $h_{\min}$ and a moderate instant show confidence τ_h. Further, the vehicle tracks are typically also quite regular meaning that the Kalman filter model works reasonably well, thereby allowing us to choose more aggressive values for the maximum track age $a_{\max}$ and the number of predicted frames n_f. Contrarily, traffic sign detections are noticeably less accurate and mix-ups with lights or specific reflections are not uncommon. However, as signs are not quite as important to detect immediately we can try to balance out these problems with more conservative tracker settings, i.e. higher

$h_{\min}$ and τ_h. Finally, the detector seems to have trouble with the extremely hard to make out pedestrians included in the CVL data. Interestingly, there are hardly any false positives but several persons are only detected very late or not at all and even then only with low to moderate confidence. This is the reason why we use relatively low τ_s and τ_h thresholds there.

It must be noted that all these parameters were determined manually through careful inspection of the results on (labeled and unlabeled) CVL videos instead of an automated search over hundreds of different configurations optimizing the final scoring metrics. This is important to not overfit the annotated CVL data. Therefore, some of these settings may only improve the perceived qualitative results rather than driving up the numeric metrics. All in all, using a number of SORT$^+$ instances specifically tuned to augment the detector's behavior for different object classes turned out to yield a solid overall tracking system.

3.6.3 Ensuring High Performance

Before concluding this chapter, we want to add a few more words about ensuring that the tracking tool is running fast enough.

While the application of the YOLOv5 model is pretty quick by itself, one must be careful not to accidentally slow things down with inefficient pre- or post-processing. For preprocessing this is ensured by paying attention to two things: a) doing all kinds of cropping and resizing before carrying out any element-wise operations and b) trying to do as many computations as possible on the GPU. The former saves time because smaller images obviously have fewer pixels to iterate through and they also consume less memory, making them faster to transfer over to the GPU. Just like convolutions are much faster to compute on GPUs that allow massive amounts of parallelization, so are most preprocessing steps (e.g. datatype conversion or scaling). For post-processing, the main pitfall to avoid is not immediately filtering away all detections with confidence less than τ and hence wasting a lot of time in NMS. Another key point to consider is that during inference YOLOv5 delivers almost the same results when computing just with half precision, i.e. 16-bit instead of 32-bit floating point numbers. With modern GPUs and good software support through CUDA and PyTorch this often yields big additional speed-ups (as well as memory savings permitting larger batch sizes).

Finally, we utilize another observation about our specific type of data, namely that our objects of interest typically only appear in specific parts of the image. In particular, our recordings typically include a very significant portion of dark sky that is essentially irrelevant for our purpose. This means we can safely cut off significant parts of the image (in this case the top third) without losing any valid detections. The applicability of this idea is confirmed by a quick analysis of the box location distribution in the CVL data (see Figure 3.6). Consequently, we can either lower the inference time by sending on the smaller image or improve the detection quality (without any loss in performance) by increasing the resolution of the cropped frame. The former is especially important when

performing multi-scale predictions, however our final version of the tool was chosen to do the latter (see Section 3.6.2 for the underlying reasoning).

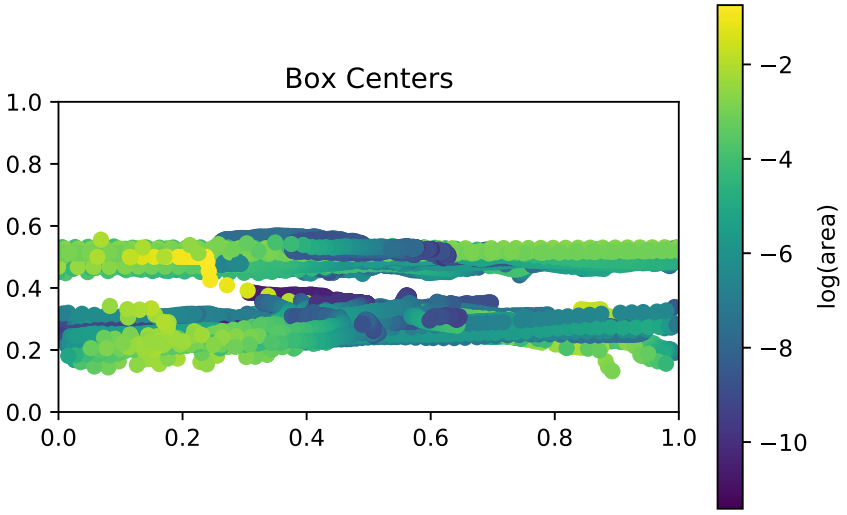


Figure 3.6: Distribution of the location of objects and their sizes in the CVL data, the x- and y-axis indicate the normalized position in the image. Note that the two prominent stripes are likely due to the videos coming from different camera setups.

CHAPTER 4

Experiments & Results

This chapter documents the results of the most interesting experiments conducted throughout this project. Furthermore, it contains the final evaluation of the eventual end-to-end tracking tool described in Section 3.6. It begins by discussing the performance of default and retrained YOLOv3 models as well as the effects of dataset filtering. Next follow studies of varying the amount of training data, learning exclusively from night images, further finetuning models already trained on car data and using bigger but slower networks. Then we explore the tracking performance of our SORT$^+$ method as well as of the state-of-the-art pure neural network tracker CenterTrack. The chapter concludes with an in-depth performance assessment of our end-to-end tracking tool, quantitatively, qualitatively and in terms of execution time.

4.1 Training Experiments

Two of the core questions underlying this thesis are how do daytime object detection models perform during the night, in particular on our dataset, and how can their performance be improved by optimally utilizing existing (although differing from our specific domain) driving datasets. To address these questions, we performed a variety of training experiments and performance evaluations. The most interesting ones (in terms of methodology and/or results) are discussed in this section. These include experiments regarding the amount of training images, training on uncorrelated versus training on correlated data, training only on night data and further finetuning of already well performing networks. The final detection model used in the end-to-end tracking tool (see Section 3.6.2) combines the results of all the experiments covered in this section.

The choice of the evaluation metrics used throughout this Section as well as their computation is explained in Section 3.3.

4.1.1 YOLOv3

As also discussed in Section 3.4.1, most of our initial experiments were conducted with YOLOv3 [RF18] as this was a state-of-the-art model at the beginning of this work, which was only superseded by more advanced methods like YOLOv5[1] during the course of this thesis. Even though our results for YOLOv5 significantly exceed those of YOLOv3, we still want to briefly discuss a few results of the latter as they provide some interesting insights that helped guiding further experiments. However, all training experiments outside of this section were performed with the better YOLOv5 model. Details about the actual implementation of all YOLOv3 experiments can be found in Section 3.4.1.

As a start, we evaluate how well default YOLOv3 trained on the general object detection dataset MS-COCO is performing on our CVL dataset. Then we study how much performance improves when training with default settings (batch-size 16, 416×416 resolution) on the full BDD-det dataset for 25 epochs (or approximately 2 days training time with ≈ 2 hours per epoch). Further, we compute detections at both 416×416 as well as 1600×1600 resolution (≈ 4 times larger in each dimension) to see if prediction quality improves when objects occupy more pixels. Figure 4.1 summarizes how the mentioned models perform on the car annotations of the CVL data in form of precision-recall curves. `default` denotes the default model, `bdd25` the BDD model and suffix `-large` indicates that the predictions were computed at 1600×1600 resolution.

First, we can see a dramatic improvement when increasing the rather small 416×416 resolution roughly 4-fold in both dimensions with the AP almost doubling in both cases (0.14 to 0.32 for the default model and 0.21 to 0.37 for the BDD model). This gap is particularly big for the default model. Most likely, this improvement is due to the fact that many of the very small cars in the CVL dataset are almost unrecognizable at only $\approx \frac{1}{20}^{\text{th}}$ of the original 1920×900 resolution (the full height of the downscaled image is only around 200 pixels due to the letterboxing described in Section 3.4.1). For the default models, the precision recall curves also drop rather sharply meaning that the models are generally very precise (almost 1.0 precision) for very confident predictions, but those only include a small fraction of all objects (as the recall falls towards 0 very quickly). This phenomenon can be confirmed by visual inspection of a few sample predictions, where it can be observed that big (obvious) cars tend to be detected very reliably but the smaller ones are hardly ever marked with a bounding box (see also Figure 4.2a). Interestingly, this behavior changes quite drastically when retraining a YOLOv3 model on the BDD dataset. Most prominently, we can see how the precision of 416×416 predictions is much lower in general and almost close to 0 for the most confident predictions. Visual inspection reveals a reason for this quite surprising behavior: the model seems to predict a lot of boxes in dark spots around the road, some of them even with confidence close to 1 (see also Figure 4.2b). This may be due to the training data containing a large number of small dark parking cars (that are basically non-existent in the CVL data), which are very hard to discern from unrelated dark spots in an image, especially in low resolution.

[1] `https://github.com/ultralytics/yolov5`, accessed: 2020-08-09

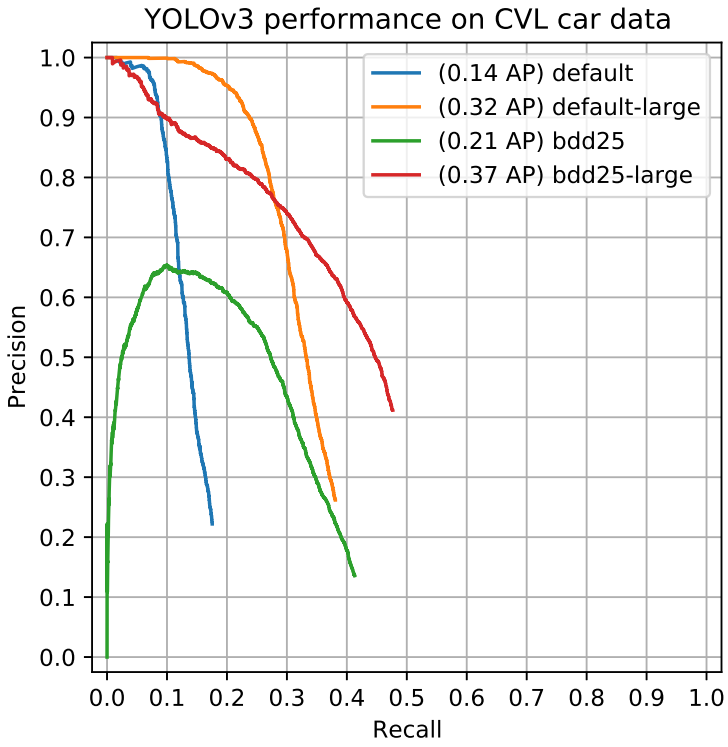


Figure 4.1: Performance summary of YOLOv3 models on CVL data.

This was one of the observations that lead to the idea of dataset filtering discussed in Section 4.1.1. The high-resolution BDD model also behaves quite differently than its COCO counterpart, namely the curve is generally a bit lower but drops only gradually, i.e. the model makes more prediction errors but also recognizes a lot more of the small objects (that the default model misses).

In summary, we can conclude that a high resolution (at least at inference time) is pivotal for good performance on our problem. Unfortunately, the detection speed drops to ≈ 4 FPS when predicting on 1600×1600 images. However, at this point we are primarily concerned with detection quality while we will deal with execution speed later. Additionally, we see that retraining on a related (although noticeably different) dataset can indeed improve performance on CVL data, which will be explored further in subsequent sections.

Dataset Filtering

The previous section demonstrated that retraining on the BDD dataset does improve performance but also leads to some unexpected behavior due to its considerate mismatch

(a) `default-large` (b) `bdd25`

Figure 4.2: Common problems observed with `default-large` and `bdd25` predictions (images cropped for better visibility). Left: Only the car closer to the camera is detected. Right: A lot of small incorrect predictions occur.

with the CVL evaluation set. Hence, one idea was to try to put the BDD-data a bit more in line with what one can typically observe in CVL videos by filtering/modifying the training labels. Section 3.2.1 describes the corresponding techniques, this section documents the most interesting respective results.

Figure 4.3 summarizes the results in form of precision-recall curves for the car class in the CVL data (similar to Figure 4.1 but of models trained on filtered datasets). On the one hand, `night10` denotes a YOLOv3 model trained for 10 epochs (the performance appeared to degrade afterwards) on a version of BDD-det that was preprocessed according to Section 3.2.1 (in particular, it contains only night and twilight data, has no boxes with high overlap and includes interesting high resolution sub-images). On the other hand, `1night` is a model trained for 50 epochs on just the night and twilight images in BDD-det combined with those of BDD-track (every third frame to avoid too similar images). The suffix `large` again indicates predictions at 1600×1600 resolution.

Once more, we can observe a drop in precision for the most confident predictions, which the models this time recover from more quickly. For low resolution predictions, the advanced filtering strategies (`night10`) appear to work roughly on par as simply training on full BDD (0.19 AP vs. 0.21 AP from the previous section), on higher ones they however significantly lag behind by almost 0.1 AP (0.28 vs. 0.37). Similarly, both results are also approximately 0.1 AP (0.19 vs. 0.30 for low and 0.28 vs. 0.38 high resolution respectively) lower than those with the less advanced filtering but longer training (`1night50`). Remarkably, `1night50`'s performance for 416×416 predictions is almost 0.1 AP better (0.30 vs. 0.21) than when training on full BDD (`bdd25` previous section). While the green curve in Figure 4.3 seems to fall off a little more quickly than the one in Figure 4.1, its average precision level is considerably higher and the precision fall-off for high confidence predictions is much less pronounced. The same is, however, not true for the higher resolution predictions, for which the pr-curves of `1night50-large` and `bdd25-large` look quite similar.

We can conclude that while the suggested intelligent filtering strategies indeed appear to

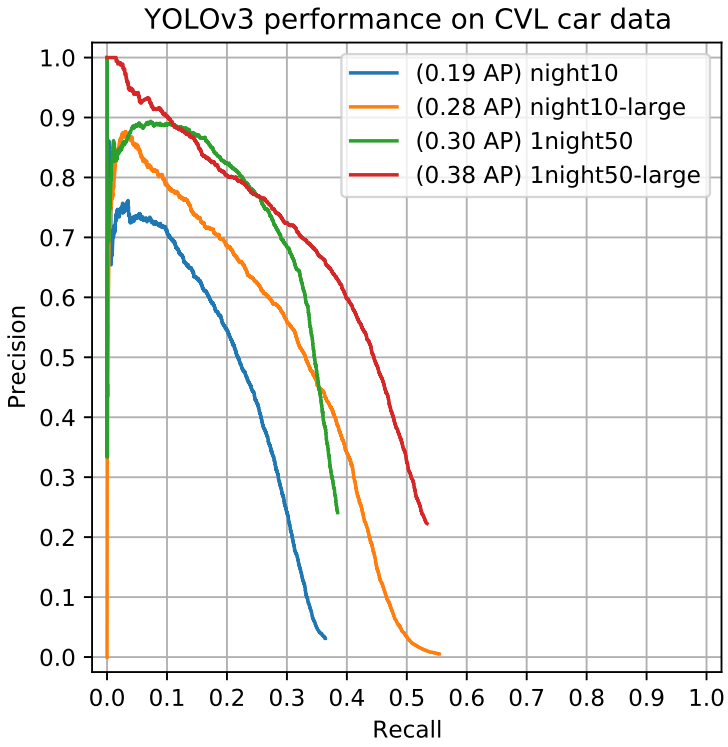


Figure 4.3: CVL data performance summary for YOLOv3 models trained on filtered datasets.

help in addressing some of the problems caused by dataset discrepancies, they do not seem to lead to better overall performance. We also observed that longer training made results worse as the model started to overfit the imperfect (due to the size of BDD we only tested relatively simple heuristic techniques that give reasonable results most of the time but of course not on every single image) filtering strategies. Furthermore, training just on night data seemed to improve results at lower resolution, but there was almost no difference over training also with day images at higher image resolution.

This concludes our coverage of YOLOv3 experiments. From this point onward, all experiments are conducted using the better performing YOLOv5 model.

4.1.2 Size of the Training-Set

Another question we looked at are how model performance scales with respect to the amount of traffic data used for retraining, which is relevant for two main reasons. First, it should give an idea about how much gains to expect from using other driving datasets (e.g. Waymo or nuScenes) in addition to BDD for retraining, before having to perform the

extremely slow process of fitting a model on almost $3 \cdot 10^6$ images. Second, it should also provide an indication of how much labeled data would be necessary to retrain exclusively on CVL data.

Our experiments regarding the influence of the training-set size are set up as follows: We train several YOLOv5 models (with settings explained in 3.4.3) for 50 epochs on random subsets of BDD-det (including daytime images) respectively containing a $\frac{1}{2^i}^{\text{th}}$ (for $i = 0, 1, 2, 3, 4$) fraction of all images. We monitor the mAP@0.5 on the BDD-det validation set after every training epoch to study the learning behavior on the differently sized training sets. In the end, we evaluate all models on our standard CVL evaluation dataset to see how discrepancies in the mAP on the BDD-det validation set correlate with CVL performance. Figure 4.4 displays the evolution of the mAP when training the models with differently sized BDD training sets. Since one epoch on a 50% smaller dataset performs 50% less weight updates, we consider the performance change not only relative to the number of epochs but also relative to (the logarithm of) the number of updates. Figure 4.5 visualizes the CVL performance of the final models as precision-recall curves. We now go on to analyze both figures in more detail.

Looking just at the final mAP results in Figure 4.4, we can see that the mAP drops only by a little more than 0.02 (0.504 vs. 0.482) even when only a quarter of the original training data is available (see the mAP values of the blue and green lines). Halving the amount of images once again (to 1/8) decreases performance more significantly, by almost 0.07 compared to the full data model. Interestingly, going from 1/8 to 1/16 does not seem to impact the mAP results any more. Considering the training curves in Figure 4.4, one can observe a temporary drop in validation mAP during the first few epochs for all but the full dataset model. This downward spike appears to be the more pronounced the less training data is available. We assume that this is caused by YOLOv5's EMA, which requires a certain number of weights updates to "burn-in" before it actually stabilizes training. Further, the noise level of the mAP-graph is higher for smaller datasets, which can be expected when training with stochastic gradient descent. While the bigger dataset models learn considerably faster with respect to the number of epochs (see Figure 4.4 top), plotting the performance increase with respect to the logarithm of the number of updates (see Figure 4.4 bottom) reveals that all the models seem to be learning at a similar pace. In fact, the graphs for the smaller dataset models (red and purple lines in Figure 4.4 bottom) do not seem to be flatting yet after 50 epochs of weight updates, suggesting that longer training may actually bring their final performance closer to the ones of bigger dataset models (at around 3000 updates, the 1/16 model even performs better than the full one).

Comparing the just discussed results with Figure 4.5 leads to the conclusion that the BDD mAP appears to correlate very well with the CVL performance. In Figure 4.5, the 1/2 and 1/4 model do only at most 0.05 AP worse than the full one while the 1/8 and 1/16 models perform roughly the same at about 0.1 less AP (than the 1/1 model). We can also see that all models achieve similar precision levels at low recall, i.e. their performance for the most confident predictions is similar, and they differ mostly in how

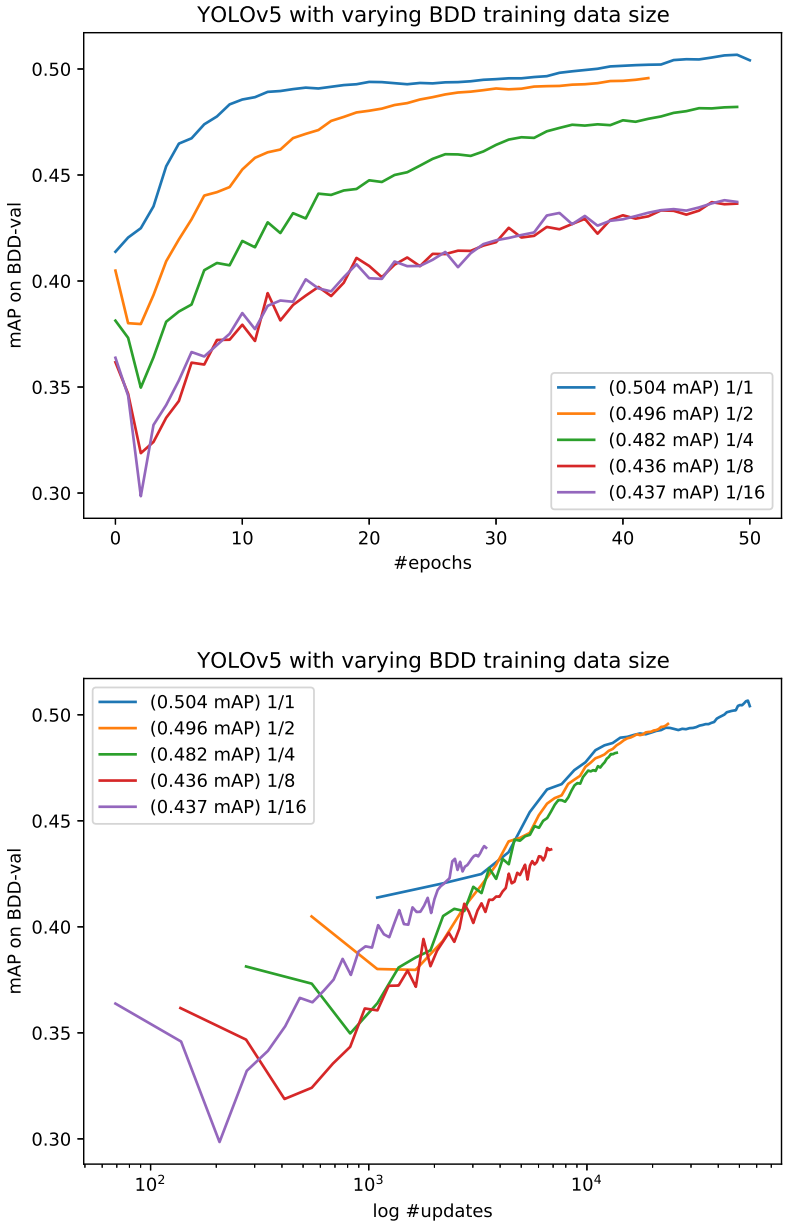


Figure 4.4: Evolution of the mAP on the BDD-det validation set when training models on differently sized subsets of BDD data, against (a) the number of epochs and (b) the log number of weight updates. ($\frac{1}{2}$ model was only trained for 42 epochs)

quickly the precision falls off as the recall is increased. This suggests that training with more data helps primarily with the more difficult and less confident detections.

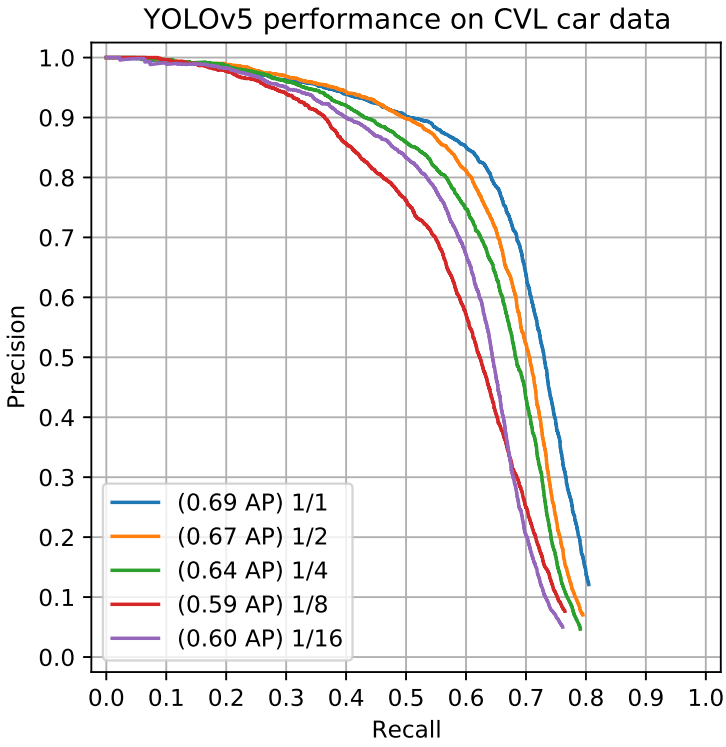


Figure 4.5: CVL performance for several YOLOv5 models trained a $1/2^i$ fraction of BDD-det images for $i = 0, 1, 2, 3, 4$. (The "1/1" model was trained for 100 epochs, the "1/2" one for 42 and all others for 50.)

Even though the performance dropping gradually as the size of the training dataset decreases is a very common phenomenon, this effect does not seem to be very pronounced in our case as the good performance (both on BDD as well as on CVL) of a model trained merely on 25% of the BDD data shows. Since training time scales linearly with the amount of data (given the same number of epochs), this means we can train a good model even with just a quarter of computational effort. Additionally, this indicates that a custom training dataset (e.g. consisting only of CVL images) must not be nearly as large as BDD-det to achieve good results. However, doubling or tripling the amount of training data by adding additional images from other large open driving datasets (e.g. Waymo, nuScenes) could result in only modest gains while dramatically increasing training time.

4.1.3 Detection vs. Tracking Data

Next, we explore how important it is that the training data is uncorrelated, by which we mean that it consists exclusively of far apart video frames that are highly dissimilar

(such as is the case for BDD-det). This is an important question as collecting and annotating large amounts of images is far less time consuming when they can all be sourced from a much smaller number of continuous video sequences (but are therefore, of course, considerably correlated). We investigate this question by training two YOLOv5 models for 50 epochs, one on 50% (to save computation time without a major drop in final performance as determined in Section 4.1.2) of the uncorrelated detection dataset BDD-det and the other one on every 8^{th} (roughly the same amount of images as for the other model) frame of the correlated tracking dataset BDD-track. The latter training dataset consists of about 25 images taken slightly more than 1.5 seconds apart from a total of 1400 videos each, whereas the former training dataset comprises 35000 images with all of them extracted from distinct video sequences. We now compare the performance of both models for the BDD-det validation (see Table 4.1) set and for CVL (see Figure 4.6).

Model	person	bike	car	motor	bus	train	truck	tlight*	tsign*	**mAP / mAP***
detection data	.65	.46	.81	.36	.58	.00	.60	.62	.71	.532 / .494
tracking data	.61	.39	.77	.31	.51	.00	.51	.00	.00	.345 / .442

Table 4.1: Comparing BDD validation performance of YOLOv5 models trained for 50 epochs on $\approx 50\%$ of BDD-det ("detection data") and on every 8^{th} frame of BDD-track ("tracking data"). The table shows the individual APs per class as well as the overall mAP. (* not included in BDD-track training data, corresponding mAP* computed without those classes.)

At first glance, we observe a clear (almost 0.2) decrease in final mAP when training on tracking data (compared to training on detection data). However, a closer look reveals that this is due to 0.00 scores for tlights and tsigns, which are not annotated in BDD-track (hence we cannot expect the corresponding model to detect them). When those classes are excluded from the mAP calculation, the difference between the results on detection data and tracking data is reduced to around 0.05 mAP*. The per-class AP differences appear fairly balanced, being ≈ 0.04 for the most frequent classes car and person and a bit higher for the others. We observe that the performance does not drop off disproportionately for any single class. Surprisingly, both models appear to perform almost the same on CVL, as shown by the AP values in Figure 4.6. This is in contrast to the results in Section 4.1.2, where we observed that differences in mAP on BDD correlate with performance differences on CVL. This suggests that those differences on CVL may rather be attributed to the varying number of update steps than the quantity of training data (as the tracking data model here uses approximately the same number of training images as the detection data one). The fact that a 0.04 car AP difference (Table 4.1) on BDD hardly affects the performance on CVL further emphasizes the domain difference between the two datasets as the AP change on BDD must stem from car types in BDD (e.g. long lanes of parking cars) that are essentially not present in CVL.

The main takeaways from this experiment are that training on correlated (but not

extremely correlated, e.g. ≈ 1.5 seconds apart) images results in noticeably worse detection performance, but the difference is smaller than one might initially have expected. Further, the CVL performance of the respective models is almost the same. On the one hand, this makes collecting a custom training dataset to achieve decent performance easier (as also mentioned at the start of this section) since the images do not have to be totally uncorrelated. On the other hand, such a custom dataset will most likely be necessary to further improve CVL performance as a significant car AP difference on BDD seemingly does not always translate to CVL.

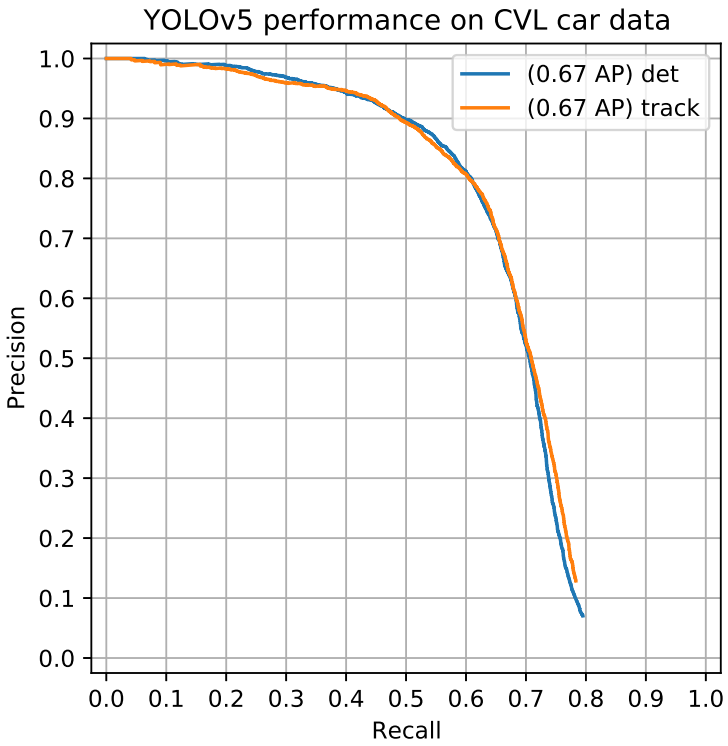


Figure 4.6: Comparing CVL performance of a YOLOv5 model trained on data from BDD-track (track) with one trained on a similar amount of data from BDD-det (det).

4.1.4 Only Night Data

So far, most experiments (except those in Section 4.1.1) were performed on mixed, i.e. containing night as well as day images, BDD data. Since our main focus is the CVL dataset with only night images, an obvious question is whether training only on the night (and the small twilight) portion of the BDD data may improve results on CVL. We study this by training another YOLOv5 model (50 epochs) on just the night and twilight images (≈ 50% of all data) in BDD-det. The performance of this model is then

compared with a model trained on a similar amount of mixed BDD-data ("mixed 1/2") and another one trained on all BDD data ("mixed 1/1") (both models were already considered in prior sections). We analyze the performance on the BDD validation set ("mixed"), only its night images ("night") and the familiar CVL evaluation data. The results are provided in Table 4.2 and Figure 4.7.

Model	**val**	person	bike	car	motor	bus	train	truck	tlight	tsign	**mAP**
mixed 1/1	mixed	.65	.46	.81	.39	.51	.00	.52	.64	.72	.523
mixed 1/2	mixed	.65	.46	.81	.36	.58	.00	.60	.62	.71	.532
night	mixed	.62	.42	.80	.33	.52	.00	.54	.60	.69	.502
mixed 1/1	night	.59	.41	.77	.37	.48	.00	.048	.57	.72	.489
mixed 1/2	night	.59	.44	.77	.28	.54	.00	.56	.54	.71	.493
night	night	.59	.42	.77	.32	.50	.00	.53	.56	.71	.489

Table 4.2: CVL performance for a YOLOv5 model trained only on night/twilight data (night) compared with other models trained on full (mixed) BDD data and 1/2 of it (roughly the same amount as there are night images).

To start with, we see that all three models perform very similarly (in terms of mAP) on the night subset of the BDD validation set. This suggests that YOLOv5 models have enough capacity to fit both night and day scenarios, hence just focusing on the former is not very helpful. This observation matches with our findings regarding dataset filtering described in Section 4.1.1. However, it also suggests that day images do not really help to improve performance during the night. Thus, they could be omitted (if one is really only interested in using the model at night) thereby speeding up training by a factor of 2. Quite surprisingly, the model trained exclusively on night data performs only slightly worse (.03 mAP) than the mixed models on the full validation set with $\approx 50\%$ daytime images. It seems that robust nighttime features generalize considerably better to daytime than one would intuitively expect. However, it needs to be noted that the initial model had been pretrained on MS-COCO, which also contains daytime cars (0.64 AP on car class). This is most certainly also a contributing factor to the night model's unexpectedly good daytime performance. The results on CVL (Figure 4.7) are in line with our expectations. All three models perform similarly well with the night only model being on par with the full BDD model but slightly ahead of the mixed model with the same amount of training data.

In summary, training just with night data seems to be neither helpful nor harmful for our use case. We found that doing so does not lead to better nighttime performance, but adding daytime images does not yield any noticeable improvements either. Interestingly, we observed that the model trained only on night data also generalizes well to daytime images.

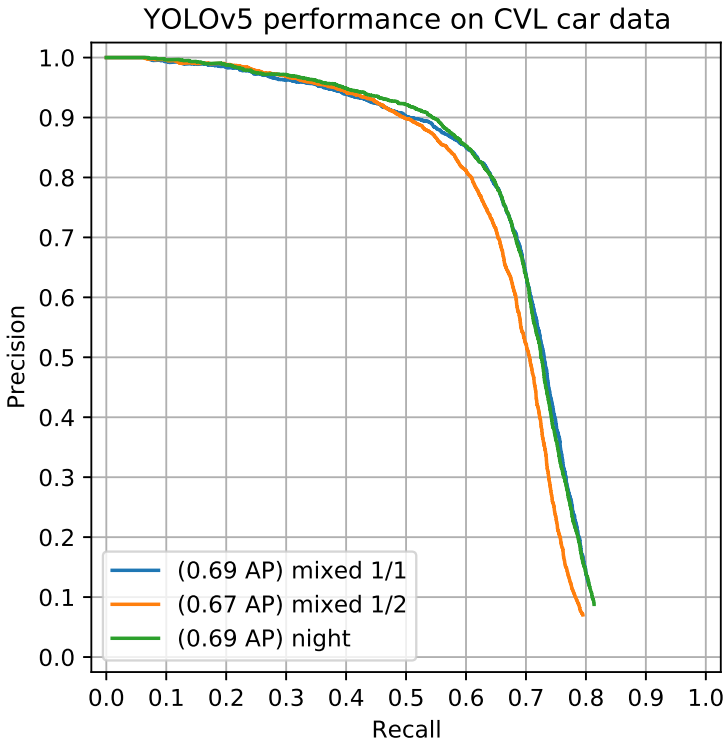


Figure 4.7: CVL performance of a YOLOv5 model trained only on night data ("night"), one trained on the full BDD training set ("mixed 1/1") and another one trained on half of it ("mixed 1/2").

4.1.5 Finetuning

Up to this point we have primarily trained models using (subsets of) the BDD dataset and with YOLOv5's default image resolution of 640×640. However, as we saw in Section 4.1.1, computing predictions on a higher resolution can lead to considerable performance improvements on CVL data, at least for YOLOv3 models. Similarly, we analyze the impact of inference resolution for YOLOv5 models in this section. Generally, one would expect a model to perform best on the image resolution it has been trained on. Unfortunately, training on large images is a lot slower, not only because every network evaluation is more expensive but also because storing gradients requires more GPU memory, hence forcing us to decrease the batch-size. Further, the training process often tends to be more noisy. Thus, we do the main training at 640×640. However, finetuning (retraining with a lower learning rate for just a few epochs) an already well performing BDD model at full (i.e. 1280×1280) resolution is a feasible alternative that we also explore in this section.

More concretely, we take our standard full BDD-det YOLOv5 model (denoted by "std") trained for 100 epochs, drop the learning rate by a factor of 100 (otherwise, the validation performance decreases significantly in early finetuning epochs) and finetune for 25 epochs on the same dataset, but this time with full resolution. We refer to this model as "fine". Then we assess the performance of both models at multiple resolutions for the BDD-validation set (see Table 4.3) as well as the CVL data (see Figure 4.8).

Model	person	bike	car	motor	bus	train	truck	tlight	tsign	**mAP**
std @ 640×	.58	.46	.75	.41	.60	.00	.61	.59	.65	.516
std @ 1280×	.65	.46	.81	.39	.51	.00	.52	.64	.72	.523
fine @ 1920×	.70	.54	.83	.49	.63	.00	.66	.71	.75	.590

Table 4.3: Performance comparison of BDD YOLOv5 model "std" at different resolutions with a version finetuned for 25 more epochs on 1280×1280 images ("fine").

Viewing the first two rows of Table 4.3, we observe that the overall BDD mAP improves only slightly when running the std-model at a higher resolution. This is despite the AP for individual classes changing very significantly. On the one hand, the high-resolution model does a noticeably better job at detecting persons, cars, traffic lights and traffic signs. On the other hand, it performs clearly worse with respect to buses and trucks. Notice, that the former are generally smaller objects that get easier to recognize at higher resolutions while the latter are typically larger objects which become harder to recognize as the model might not have learned to deal with objects of the increased sizes. We would expect additional higher resolution finetuning to fix this defect, which was confirmed by our experiments. The performance on all classes is improved, for some (such as bikes and traffic lights) very significantly. Further, the finetuned model even reaches slightly higher APs for trucks and cars than "std @ 1280". Overall, this leads to a 0.07 higher total mAP at no additional inference cost (just longer training times), a good improvement.

Next, we study how high-resolution finetuning affects CVL performance, visualized in Figure 4.8. Just like for YOLOv3 (see Section 4.1.1), we also observe a gap between the quality of the 640- (indicated by "small") and 1280-predictions ("medium") produced by YOLOv5. However, the car AP is now almost twice as high as the one achieved by the YOLOv3 models (Figure 4.1) and YOLOv5 can comfortably run in real-time at 1280×640 resolution (CVL images have a roughly $2:1$ aspect ratio as discussed in 3.4.1). Increasing the resolution even further to 1920×1080, however, does not seem to improve performance, neither for the standard nor for the finetuned model. Perhaps this is related to the full resolution of the BDD training images being just 1280×720. Although high-resolution finetuning leads to very significant improvements on BDD, those unfortunately do not seem to translate over to CVL. In fact, the CVL performance even drops slightly. However, it needs to be noted that CVL data only have (sufficiently many for a meaningful evaluation) annotations for cars and the car AP for BDD only changed by 0.02 due to finetuning (further, we already observed that even bigger BDD

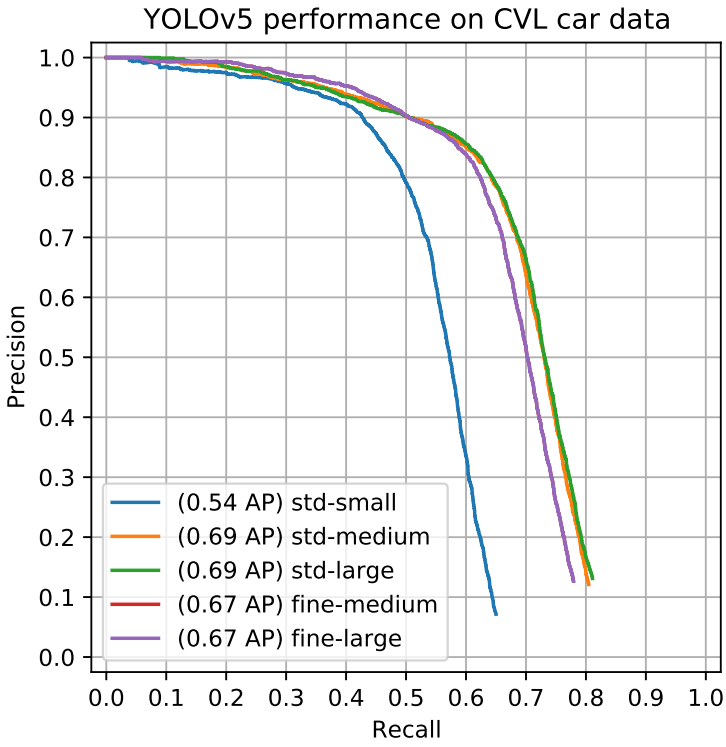


Figure 4.8: CVL performance of BDD YOLOv5 models at different resolutions and with/without finetuning.

car improvements hardly affect CVL performance in Section 4.1.4). Despite the minor drop in CVL performance, we nevertheless select exactly this finetuned model as the object detector in the final end-to-end tracking tool as it produces noticeably better detections for other classes.

Before we conclude this section, we want to briefly mention another finetuning related experiment, that is finetuning on some CVL data. In particular, there also exists the so called *sparse-CVL* dataset, a collection of around 500 annotated images extracted from CVL videos with sufficient temporal spacing to ensure minimal correlation. We then finetune on this dataset for 25 epochs in exactly the same manner as for the higher resolution model. The hope was that this would allow the rather general BDD model to adapt more closely to our particular domain. Unfortunately, the training process was quite unstable and the eventual detection performance seemed to drop rather than to rise. A more detailed inspection of the sparse-CVL data revealed that most of the images are actually not very representative of the CVL validation data. For example, there are very few scenes of small far away cars that are prevalent in the validation data (and several of the finetuning images were daytime recordings). Consequently, finetuning on sparse-CVL

is unlikely to improve results and thus the idea was abandoned rather quickly. However, we still believe that finetuning on a small but actually representative set of CVL data is a promising direction for future work.

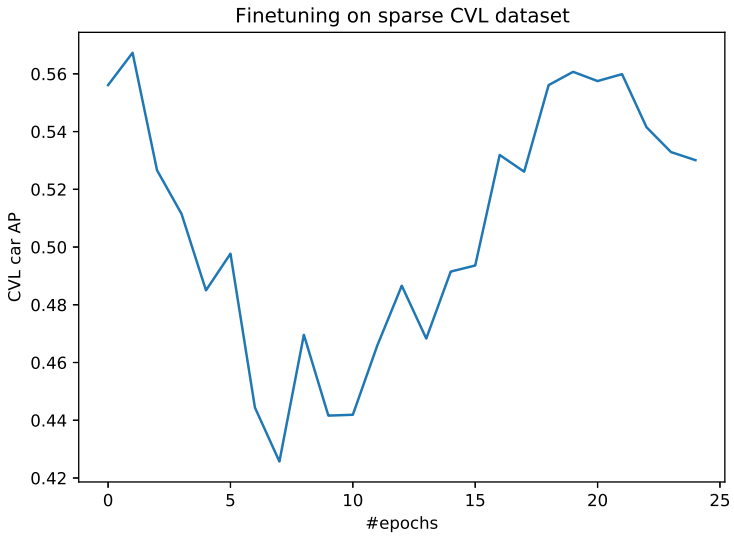


Figure 4.9: Training behavior when finetuning BDD YOLOv5 model on a small sparsely annotated CVL dataset.

In summary, we found that finetuning a BDD model trained at 640×640 for a few more epochs at 1280×1280 significantly improves performance when computing detections at the latter resolution, especially on typically rather big objects. In contrast, trying to finetune on a small set of CVL data was less successful, which we however suspect to be due to this particular dataset not being representative enough of our target domain.

4.1.6 Increasing the Model Size

Lastly, we wanted to see if, disregarding inference speed, a bigger model could reach even better detection performance than the finetuned YOLOv5-M model predicting at 1280×640. For that purpose we trained a YOLOv5-L model with exactly the same training procedure, i.e. 100 epochs at 640×640 followed by finetuning for 10 (we stop already after 10 as this process is very time consuming and no more noticeable improvements can be observed) epochs at 1280×1280 with reduced learning rate. Figure 4.10 shows the evolution of the validation mAP during the training of both the familiar YOLOv5-M and the new YOLOv5-L model.

First, we can spot (in Figure 4.10) that the L-model has an approximately 0.03 higher mAP after the very first epoch. This gap shrinks slightly over the next few epochs and is then maintained at 0.02 over the entire first phase of training. At the end, the L-model

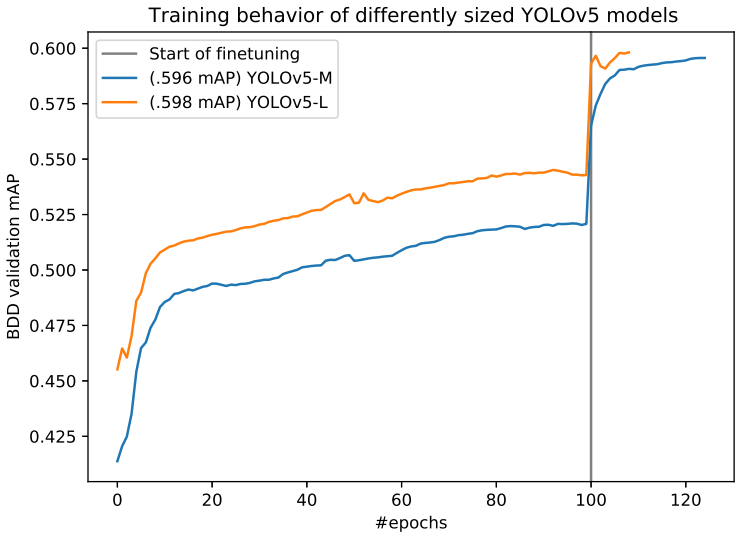


Figure 4.10: Comparison of training and finetuning (on higher resolution) a YOLOv5-M and YOLOv5-L model on BDD.

performs about 0.02 AP better at 640×640 resolution. The small oscillation of the L-model's graph at around 50 epochs is caused by restarting training and thereby renewed EMA burn-in (see also Section 3.4.3). As soon as finetuning starts, the mAP gap between the two models shrinks rapidly. While the M-model is continuously improving, the L-model's performance dips briefly (after epoch 100), then recovers but plateaus quickly (hence we terminated the slow training process already after 10 epochs). Eventually, at the end of training the L-model is only 0.002 (0.598 vs. 0.596) better than the M-one at 1280×1280 but is approximately 33.3% slower to execute. Since these initial results were not very promising (and the L-model too slow for real-time performance anyways), we did not further explore the direction of increasing model size.

4.2 Tracker Experiments

After extensively exploring how to train an effective detection model for our particular use-case in Section 4.1, we now experiment with extending this model to a full tracking method. Since CVL tracking performance is primarily dictated by the quality of the detection model (as will become evident in Section 4.2.1), this coverage is kept comparatively brief. We compare the performance of our SORT$^+$ tracker (see Section 3.5) with several baselines and discuss the impact of the evaluation IoU threshold on the calculated metrics. Finally, we also test CenterTrack [ZKK20], a state-of-the-art tracking technique that has been particularly successful on driving related tasks.

4.2.1 SORT$^+$

In this section, we evaluate the effectiveness of our proposed tracker by comparing it to several baselines. Note that we only consider aggregate metrics here, a more fine-grained analysis follows in Section 4.3.

As a first baseline, we consider a general purpose YOLOv3 [RF18] object detection model trained on MS-COCO (which also includes cars) in combination with a basic SORT [BGO$^+$16] tracker. YOLOv3 was a state-of-the-art detection model when the project started and SORT is a simple but effective (especially for our particular use-case as discussed in Section 3.5) technique of turning a detector into a full tracker. We evaluate YOLOv3 at a resolution of 416×416 to ensure that this baseline is running in real-time like the other methods that are about to follow. Next, we look at SORT based on our detection model (see Section 3.6.2) resulting from the extensive training experiments in Section 4.1. Finally, we evaluate our detection model in combination with our SORT$^+$ (see Section 3.5) extensions to basic SORT. We compute the set of metrics defined in Section 3.3 with a box matching IoU of 0.5; the respective results are given by Table 4.4.

Method	pre [%]	rec [%]	ids [#]	MOTA [%]
Real-time baseline	30.8	14.6	83	-18.6
Our detector + SORT	**89.5**	50.7	48	44.3
Our detector + SORT$^+$	80.0	**61.3**	**16**	**45.8**

Table 4.4: Comparison of CVL tracking performance for different methods at 0.5 IoU. Precision (pre), recall (rec), number of ID-switches (ids) and MOTA are shown. The best score of each column is written in bold.

Going through this table line-by-line, we immediately see that the initial baseline performs rather badly, reaching only $\approx 30\%$ precision, 15% recall and even a negative MOTA. Further, the number of ID-switches is almost twice as large as the one of the second best method. Clearly, this model is far away from being practical. Next, we see how using our YOLOv5 model (retrained on BDD and finetuned on higher resolution) dramatically improves all scores, precision by a factor of ≈ 3, recall by a factor of ≈ 3.5, the number of ID-switches is almost halved and the MOTA jumps by over 60%. These are all already good scores and this tracker is, albeit still far from perfect, definitively useful in practice. Perhaps the most concerning aspect of this method is that it only detects about half of all annotated objects ($\approx 50\%$ recall). As discussed in more detail in Section 3.5, this is due to basic SORT being quite conservative when outputting predictions, which was exactly the reason why we developed the SORT$^+$ techniques. As can be seen in the last line of the table, using SORT$^+$ improves the recall by more than 10%, although this comes at the cost of a similarly lower precision and thus only a minor increase in summarizing MOTA. Further, the number of ID-switches is reduced by a factor of 3 meaning that tracks are maintained far more consistently, which was another goal of the SORT$^+$ enhancements (and is, in general, a very significant improvement even though it hardly affects the MOTA).

While the numeric results discussed so far are already decent, a careful manual analysis of the resulting videos gave the impression that the results may be even better than the raw numbers would suggest. In fact, we realized that quite frequently objects would be correctly tracked but their bounding boxes would not have high enough IoU with the ground truth, meaning that those correct (but not super accurate) detections were counted as errors. Figure 4.11 contains two examples of such situations. This effect is further amplified by imperfect (sometimes even inconsistent) annotations, in particular also when the true bounding box is ambiguous, e.g. when a part of a car is too dark/blurry or hidden in some other way. Therefore and because we first and foremost care about tracking cars at all, where the accuracy of the bounding box is only secondary, we also computed all evaluation metrics with the lower IoU threshold of 0.25. Those results can be found in Table 4.5. In this setup, the precision and recall increase by 8 and 7 percent, respectively, while the MOTA correspondingly improves by almost 15%. In addition, the recall and MOTA gap between our detector plus SORT and our detector plus SORT$^+$ is also much more significant. We believe that these numbers are considerably more representative of the perceived performance when watching the output videos and thus continue further evaluations with the 0.25 IoU threshold.

Figure 4.11: Tracking situation examples where a car is correctly detected (the color just indicates the object identity while tracking) but the IoU of the predicted box with the true bounding box (which is also a inherently a bit ambiguous in the left picture) is rather low. Images were cropped for better visibility.

Method	pre [%]	rec [%]	ids [#]	MOTA [%]
Real-time baseline	47.5	22.5	104	-3.3
Our detector + SORT	**95.9**	54.3	59	51.5
Our detector + SORT$^+$	88.8	**68.1**	**19**	**59.3**

Table 4.5: Comparison of CVL tracking performance for different methods at 0.25 IoU. Precision, recall, number of ID switches and MOTA are shown. The best score of each column is written in bold.

All in all, we can conclude that a simple tracking baseline is clearly not sufficient to solve our problem. However, dramatic improvements can be achieved with better techniques, ultimately leading to a certainly practical solution. The experiments in this section also highlight that the tracking performance in our particular setting is foremost influenced by the quality of the object detection model. Nevertheless, further noticeable performance increases (assuming representative evaluation) can be achieved with tracking enhancements designed specifically with our use-case in mind.

4.2.2 CenterTrack

While work on this thesis was progressing, CenterTrack [ZKK20], a new pure neural network tracker, appeared. This model presented especially promising results on various driving related datasets. Hence, we decided to test CenterTrack on our CVL data. As already discussed in Section 3.4.2, we directly use the author's pretrained model for the nuScenes dataset (which is rather similar to BDD) and lower the tracking threshold to 0.1 to increase the method's recall. The results of applying CenterTrack to CVL at both the default 800×448 resolution and approximately double of that (in each dimension) are given by Table 4.6.

Resolution	pre [%]	rec [%]	ids [#]	MOTA [%]	FPS
800×448	68.5	37.5	34	19.9	20
1600×800	81.7	36.1	40	27.9	7

Table 4.6: CVL tracking performance of CenterTrack at 0.25 IoU for two different input resolutions.

Although CenterTrack is as expected much better than the YOLOv3 COCO baseline we looked at in the previous section, its MOTA is less than half of the one of our tracker, as can be seen from the comparison of Table 4.5 and Table 4.6. A closer analysis reveals that this discrepancy is primarily rooted in the much lower recall. Comparatively, the precision is relatively close to that of our tracker. Interestingly, it seems that increasing the prediction resolution only improves the precision but not the recall, which differs from what we observed throughout the majority of our experiments in 4.1. Further, the CenterTrack output contains almost twice the amount of ID-switches as the one of our tracker. This may be explained by to locality of CenterTrack, i.e. because it only tracks from one frame to the next. Studying some of the output videos indicates that CenterTrack tends to struggle with small cars. This may be related to the fact that CenterTrack can only learn to detect a single object in every grid-cell. Another explanation would be that it is rooted in the nature of the training dataset. Given CenterTrack's excellent results on other driving related datasets, we suspect that CVL performance could also be improved by retraining on BDD (which leads to good results for our other models) and tweaking of various tracking related hyper-parameters. Whether this would be sufficient to fully bridge the performance gap to our tracker is, however, uncertain. Due to retraining CenterTrack being very resource intensive, we leave this

question for future work. Finally, it needs to be noted that the higher resolution version runs only at 7 FPS, which is quite far away from our real-time requirement.

To gain a few more insights into CenterTrack's CVL behavior, we also plot its precision-recall curve and compare it with the one of our tracker (see Figure 4.12). Note that we only consider boxes returned by the tracker, hence the curves end at significantly higher precision levels than the pr-curves previously shown in this thesis. Consequently, the AP is also lower. While lowering the tracking threshold even more to beyond 0.1 may result in the CenterTrack curves extending further, they have already started to bend so strongly that big performance improvements are unlikely to be achieved in that way.

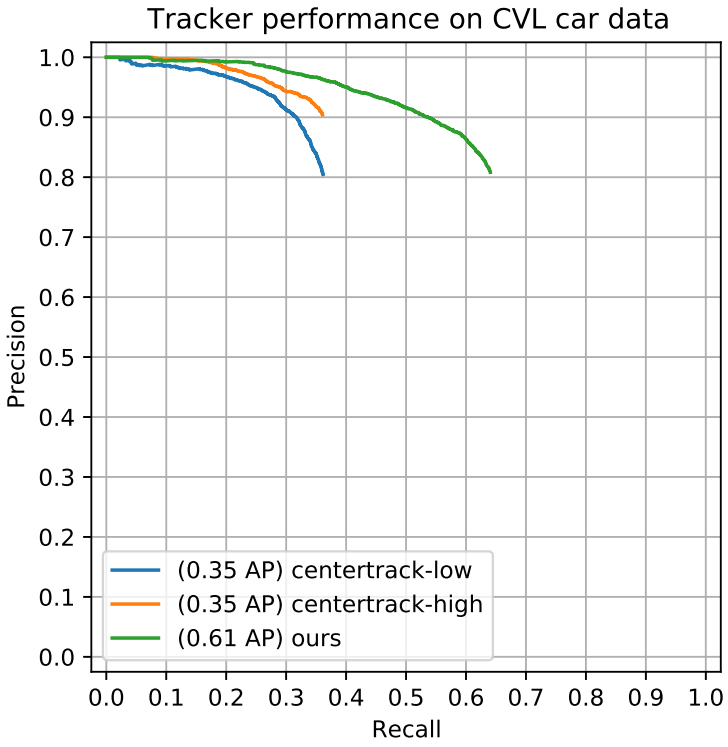


Figure 4.12: Comparing the CVL detection performance of CenterTrack at two different resolutions with our tracker.

To summarize: even though CenterTrack achieves excellent results on many driving related datasets, it does not appear to be the most suitable method for our particular problem. While CenterTrack's performance on CVL can probably be improved with further work, significantly surpassing the performance of our tracking methods seems not very likely at the moment, considering the performance gap observed in our initial tests.

4.3 Final Evaluation of Tracking Tool

In this last section of the chapter we perform a detailed analysis of our final tracking tool's performance on CVL data: First quantitatively by comparing various performance metrics, then qualitatively by manually studying the output videos and finally in terms of execution speed.

4.3.1 Quantitative Performance

We begin with a quantitative assessment of the tracking performance. Similar to Section 4.2, we compute the precision, the recall, the number of ID-switches and the MOTA for our tracker's results on the CVL evaluation dataset. This time, however, we also take a closer look at the performance on individual videos, all listed in Table 4.7.

Video	pre [%]	rec [%]	ids [#]	MOTA [%]
1	95.0	39.6	1	37.3
2	39.0	57.0	0	-32.1
3	99.8	77.3	1	77.2
4	78.9	95.7	0	70.0
5	100.0	92.5	0	92.5
6	92.7	87.3	4	80.3
7	98.3	67.9	2	66.1
8	98.9	48.3	0	47.8
9	89.6	50.5	6	44.4
10	98.5	49.2	1	48.3
11	99.3	64.7	0	64.3
12	91.0	82.1	1	73.9
13	94.6	83.6	0	78.8
14	45.8	14.1	1	-03.8
15	30.1	42.3	1	-56.4
16	98.8	99.3	0	98.1
17	100.0	48.1	1	46.3
overall	**88.8**	**68.1**	**19**	**59.3**

Table 4.7: Performance numbers for all videos. Shown are precision, recall, number of ID-switches and the MOTA. The overall results were computed as a mean over all individual scores (sum for ID-switches).

First, we can see that the precision in most cases hovers around 90%, there are just 3 outliers with significantly lower values. The CVL evaluation videos are quite diverse and depict nighttime driving situations with different lighting conditions in various environments. Hence, it is not surprising that some of the video clips are much more difficult for our tracker than others. The recall shows much stronger variation, with some videos going above 90% while several others score just around 50%. Further, we can

recognize that some videos with low precision (beyond 50%) also have rather low recall (beyond 50%). This may provide an explanation for the low precision values despite our model being generally quite accurate: if the recall is low, the model has output only few detections, hence if some of them are wrong, the precision decreases very quickly. The number of ID-switches is generally small, apart from two outliers, which can be explained by video 9 being by far the longest one and video 6 featuring an unusually dense object population. Influenced by the recall scores, the MOTA values also show considerate variation, some of them are very high (above 0.7) but a few are even negative (meaning that there are more errors than actual ground truth boxes). However, it has to be noted that some of the CVL videos are very short and contain only few objects. Consequently, if a tracker happens to have problems with some particular object instance, this can easily lead to low scores on the corresponding video.

Second, we compare the raw detections returned by our underlying neural network and the final tracker output in terms of pure detection performance (i.e. no object IDs). The corresponding precision-recall curves are visualized in Figure 4.13. We can see that the tracker's curve extends roughly to the point where the detector's precision starts to clearly fall off, indicating that the tracker is indeed configured well. While both curves match almost exactly for high confidence predictions, as we would expect, the tracker's curve slightly surpasses the detector's in the higher recall regime. This can be seen as a sign that our various corrective mechanisms are working properly and indeed improve performance (at least slightly).

Overall, while there is still considerable room for further improvement, the quantitative results are in general quite encouraging and suggest that the current tracker might already be useful in practical situations. This claim will further be verified through a careful qualitative analysis of the output videos in the following section.

4.3.2 Qualitative Performance

After evaluating the performance of our tracking tool through various numeric metrics, we now take a closer look at actual result images (videos). More concretely, we study in which situations the tool already works well and what kind of errors it still makes. At the same time, we also pay special attention to unlabeled (and thus not considered by the previous performance scores) object classes such as traffic signs. Finally, we also discuss how our tool performs on various other videos recorded throughout the CarVisionLight project, which depict special testing scenarios that differ substantially from those in our general CVL evaluation set.

Figures 4.14 and 4.15 present a variety of sample images output by our tracker. Tracked objects are marked by their bounding box with the color specifying the object identity. Additionally, the object class ('c' for car, 'ts' for traffic sign, 'p' for person and 'm' for motorbike) as well as the tool's confidence estimate are also shown. The images were selected to show a diverse set of driving situations encountered in the CVL dataset. It should be noted that the tracker performs well in all of the displayed images, failure

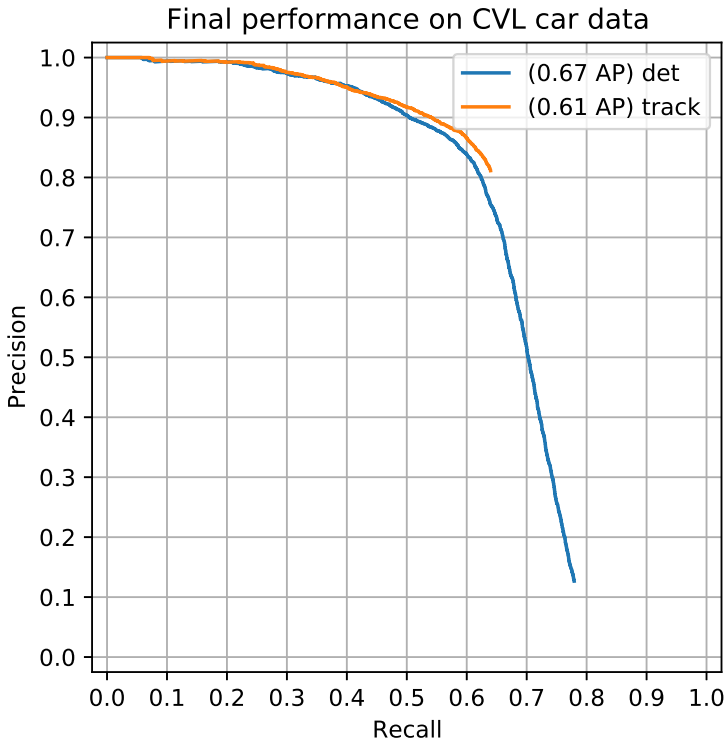


Figure 4.13: Comparing the CVL detection performance of the underlying model ("det") and the final tracking output ("track").

cases will be treated separately later in this section. Nevertheless, the performance seen here is actually quite representative of the general tracking quality in most situations.

The images provide further insight into the behavior of the tracking tool. Now follows a list of our main observations from studying the given images and the full output videos in general.

- The tool is able to recognize not only oncoming cars (see Figure 4.14 c, d, e, f) with bright head lamps but also cars driving in the same direction with comparatively subtle red rear lights (see subfigures a, b).
- Even many parking cars with no individual lighting are often tracked accurately (Figure 4.14 a).
- Strong ground reflections or other light sources such as street lamps generally do not seem to confuse the tracker too much (subfigures c, d).

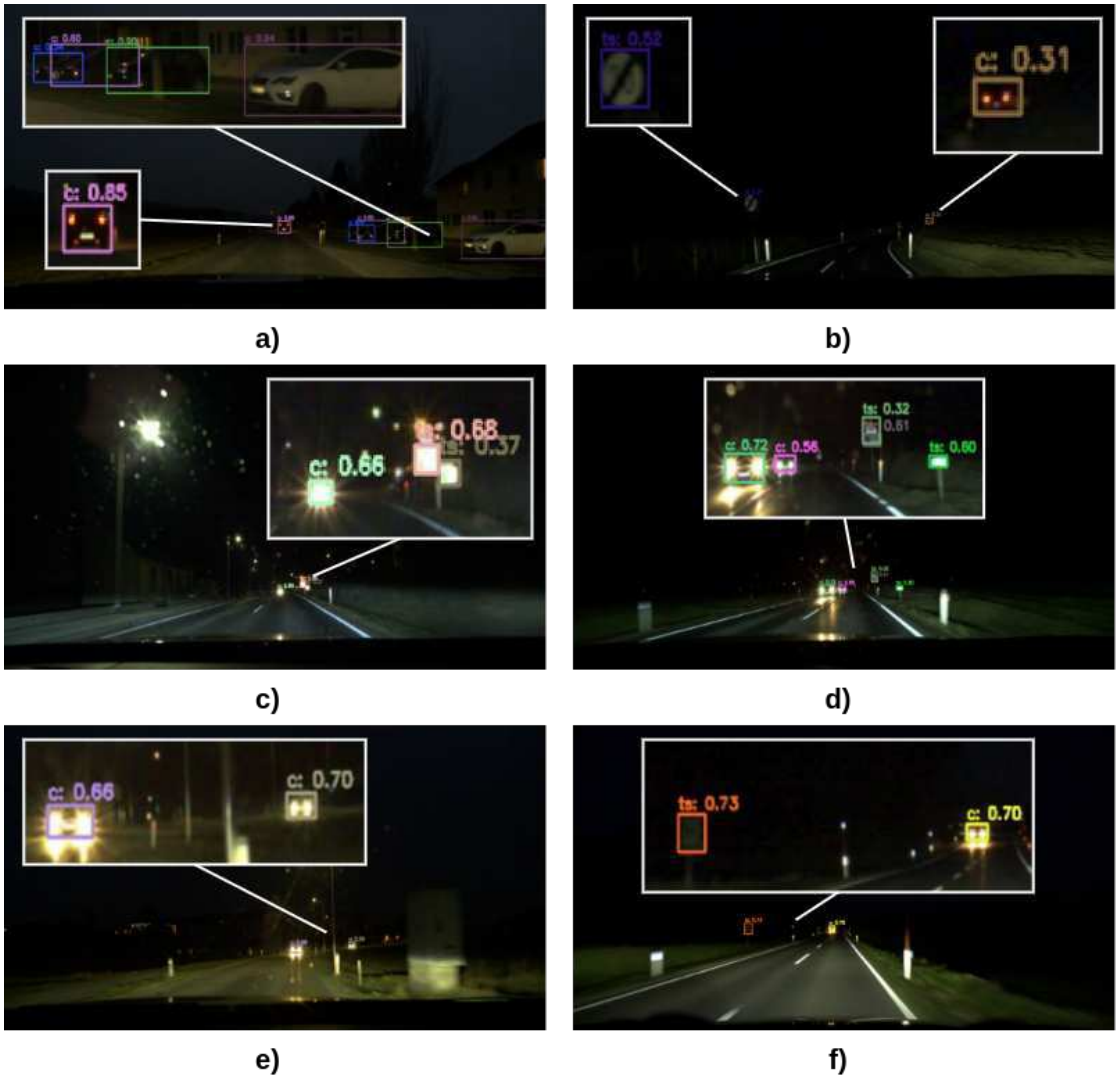


Figure 4.14: Selected tracker output examples captured from various CVL videos.

- Cars are usually already first detected when they are still quite far away and just the head lamps can be seen (see Figure 4.15 h, i, j). However, the tracker also handles closer and thereby much clearer cars (subfigure g).
- The tool tracks multiple objects simultaneously (subfigures g, h, k).
- Further, the tool seems to generalize well to various image noise levels (subfigure j) and even different cameras (subfigure i).
- In addition to cars, traffic signs are also detected quite well (subfigures h, l). Not just when they are clearly visible, but sometimes even before a human would spot them (when watching the video).

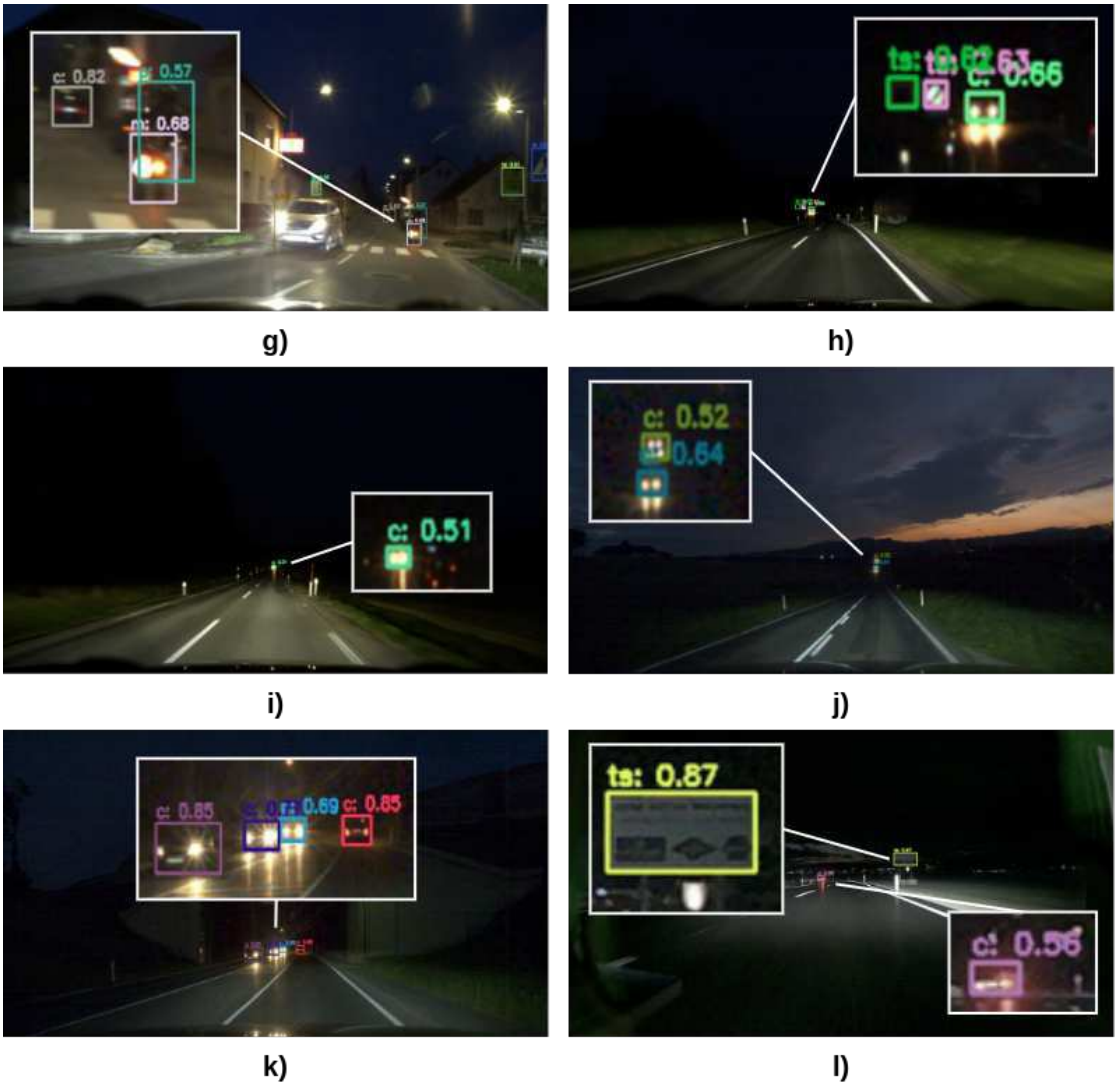


Figure 4.15: More selected tracker output examples captured from various CVL videos.

Overall, the tool appears to generally work well across many different scenarios. However, there are still occasional errors, which we discuss next. Figure 4.16 provides some examples of common failure cases. First, sometimes the tracker simply does not pick up an object. For example, the posts of the street lamps in the top left picture seem to cause quite inconsistent tracking of the car in front of the camera. Additionally, some really dark parking cars are not detected, although those particular ones (in the top left image) are difficult to spot even for humans and not very relevant for the driving car. While we found that the model rarely makes mistakes in incorrectly detecting bright spots as cars, it does sometimes misinterpret specific kinds of reflections as traffic signs, like for example in the top right image. While the model is generally very precise, it

still sometimes outputs seemingly random detections, like dark spots next to the road (detected as cars) as in the bottom left picture. Further, in some instances far-away cars are detected considerably later than a human would recognize them. This issue is especially common when the road is not straight (e.g. bottom left image). Finally, the model is not nearly as consistent at tracking traffic signs as it is for cars. Sometimes only a subset of the visible signs is detected or a previously tracked sign is suddenly lost for no apparent reason (see also the bottom right image). These are the most common types of errors we observed while extensively studying the tracker output, but they are by no means prevalent and the main tracking quality is certainly reasonable as is also confirmed by the numeric evaluation in Section 4.3.1.

Figure 4.16: Tracker output examples illustrating common failure cases. Images best viewed digitally.

Finally, we also applied our tracking tool to some other CVL videos not part of our standard evaluation set. Those scenarios were captured for controlled testing of other aspects of the CarVisionLight project. They do not necessarily represent general nighttime driving situations, but it is nevertheless interesting to see how our tracker performs in situations that have not been considered at all during development. Figure 4.17 provides several example outputs for three different video sets, the first aimed at pedestrian detection, the second at car detection in very dark situations and the third at car detection at daytime. We now discuss those results in more detail.

While pedestrians are in principle detected, it appears that they have to be quite close for that to happen reliably. Further, the tracker struggles with detecting persons in unusual situations, such as when crouching (top row, second image). We suspect that

Figure 4.17: Tracker output examples for other CVL videos considerably differing from the ones in the CVL evaluation set.

this is due to the BDD dataset containing primarily walking (and hence in up-right position) pedestrians. Further, most pedestrians in the city walk in places where it is considerably less dark than in the videos here, which could also explain the less than ideal performance. The second row in Figure 4.17 shows cars with active lamps in night scenes that are considerably darker than the ones most commonly found in the CVL evaluation set. For the most part, the tracker's good performance on cars also generalizes to those videos, at least for far and near cars. Strangely, tracks are sometimes lost at medium distance, a phenomenon we did not observe in other videos. We suspect that this is caused by a combination of similarly close cars in the BDD training set typically being already more easily recognizable due to better lighting and the noticeable camera vibrations throughout the periods of no detection. Lastly, we also took a brief look at the tracker's daytime performance. Although this project's main goal is tracking during the night, it is beneficial that the tool also works well during the day, as the displayed images confirm. As a final remark, one can see how tractors (not a training class) are quite reliably recognized as trucks. All in all, our tracker performed well on those additional videos, which are considerably different from the ones in the CVL evaluation dataset.

In conclusion, a visual inspection of the tracker output agrees with the solid evaluation scores computed in the previous section. While there is still further room for improvement, we think that the current version of the tracker could already have some real-world uses.

4.3.3 Speed

As real-time performance is an important goal of the CVL project, we finish the evaluation of our tracking with a speed test. This is done by running the tracker on the CVL evaluation dataset (plus some other CVL videos that previously had to be excluded due to missing labels) and recording the average FPS. Note that loading the images from the disk and writing out the result videos with drawn bounding boxes is not included in the timing as neither of those would be necessary in a real application. Furthermore, we run the same experiment with a simulated camera (see Section 3.6.1), i.e. only providing an image every 33 milliseconds, and count the number of skipped/dropped frames. The results are visualized in Figure 4.18.

Our tracker averages around 33 FPS across all of the tested videos. Only 3 of the 26 videos dip very slightly below the real-time mark of 30 FPS. Those are videos that are particularly crowded with objects (many cars and traffic signs), hence both non-maximum suppression as well as the matching of predictions to tracks are more computationally intensive. Overall, it can be said that the tracker comfortably fulfills the real-time requirement. It also needs to be noted that being much faster than 30 FPS would be a waste as additional computation time could be better spent on techniques that enhance detection/tracking quality. Analyzing the number of dropped frames when using our camera simulation shows an interesting phenomenon, namely that the tracker drops at least 5 frames in every single video. Additional analysis revealed that a single drop of 5 consecutive frames always happens close to the beginning of every video, presumably caused by some not explicitly controlled initialization in the YOLOv5/SORT library code. As this could be easily resolved in a real application by giving the tracker some time to setup, these frame drops can be disregarded for our purpose (note that if the videos are very short, these always dropped frames may also contribute to the few videos being sub 30 average FPS). Just a few videos skip additional frames, with only one of the longest CVL videos ones dropping more than a single one.

All in all, the speed of our tracking tool is clearly satisfactory. It is fast enough to run in real-time while using available extra performance for techniques to ensure maximum detection and tracking accuracy.

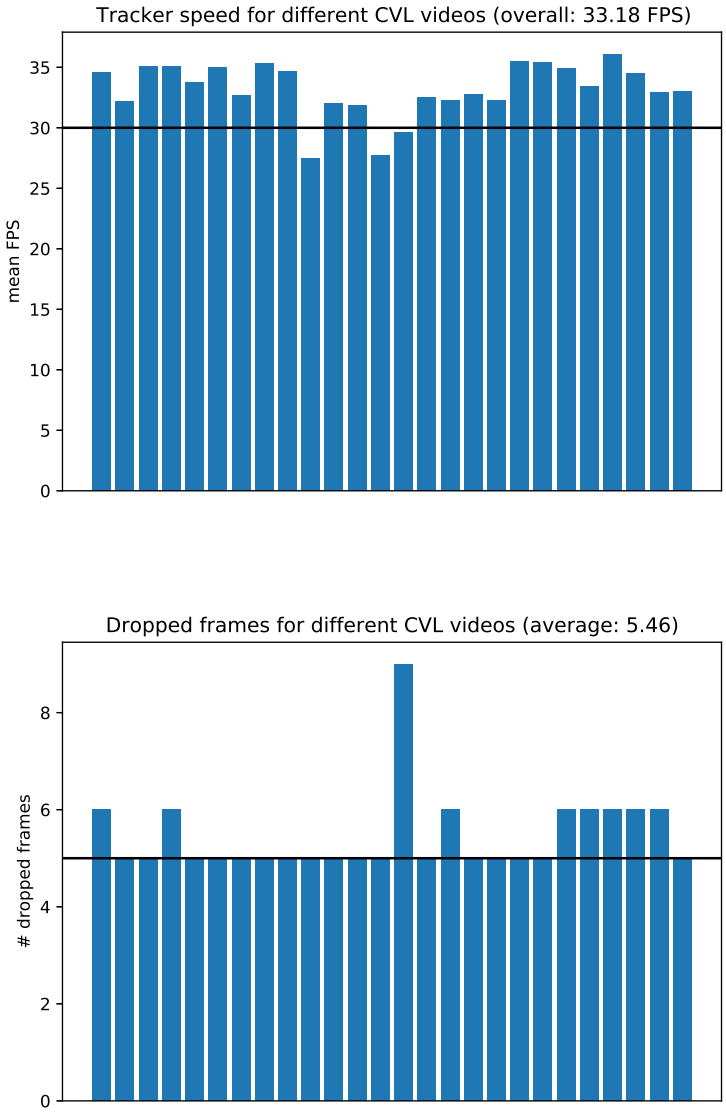


Figure 4.18: Results of the tracker speed test with the mean FPS per video (top) and the number of skipped frames when simulating a real camera (bottom). In the bottom plot, the black horizontal line indicates the number of frames that are always dropped at the beginning (which could be avoided by giving the tracker time to set up).

CHAPTER 5

Conclusion & Future Work

This final chapter gives a summary of all parts of this thesis together with corresponding individual and overall conclusions. Lastly, we outline several potential future research directions.

5.1 Summary & Conclusion

In Chapter 1 we discussed how an advanced nighttime driver assistance system could present a significant step forward towards safer driving under difficult conditions. Further, we defined our problem of 2D multiple object tracking in RGB-camera videos and argued its importance. Finally, we stated our research goals, which can be broadly summarized as exploring techniques for adapting modern daytime object detection and tracking methods to also work well at nighttime, in particular on the special dataset collected throughout the CarVisionLight project.

Chapter 2 surveyed the current state-of-the-art in object detection, object tracking and further research areas related to nighttime vision. We first described early methods that tracked cars solely based on detecting the lamps as bright spots in the images, discussed their shortcomings and motivated the use of deep convolutional neural networks. Next, we gave an overview of deep learning based object detection methods, explaining fundamentals but also covering advanced ideas commonly found in modern models. Then we discussed techniques for extending detection networks to full tracking methods that handle temporal association, as well as recent approaches for doing so with just a single neural network. We also mentioned relevant driving related datasets and listed their most important characteristics. Finally, we briefly touched upon other interesting ideas related to adapting daytime models to work well during the night. With several highly relevant papers being published as work on this project was progressing, it is clear that object detection and tracking are still very active research areas. Additionally, more

publications are also starting to consider the problem of vision in unfavorable settings such as during the night, indicating that our work was indeed following a current trend.

Chapter 3 dealt with important implementation related aspects of this thesis. We justified our general development environment and discussed fundamental design decisions such as the common data layout and our particular evaluation metrics. Then we documented which open-source projects we built our work upon and how we configured and modified them. Next, we introduced several enhancements to the basic SORT tracking algorithm specifically designed to improve tracking performance in our particular domain. We denote this improved version of SORT by SORT$^+$. Finally, we provided a detailed description of our end-to-end tracking tool that combines all of our insights into a self-contained tracking application. We discussed critical implementation aspects such as automatically cropping, resizing and rescaling input images to ensure optimal speed and detection performance. Then we described our procedure for utilizing the Berkeley Deep Drive (BDD) dataset to train a powerful YOLOv5 object detection model. Finally, we explained how using several tracking levels, differently configured SORT$^+$ instances, allows us to account for class specific behavior of the detection model, thus improving overall performance. To summarize, this chapter covered how to build an effective tracking system optimized specifically to work well on the CarVisionLight (CVL) dataset.

Finally, in Chapter 4 we documented various experiments about how to favorably use the existing BDD training data for training effective models that perform well on CVL evaluation data. We studied how the size of the training set affects the eventual model performance and found that good results can be reached even with just 25% of all BDD data and respectively also 25% of computational effort. Further, we discovered that training exclusively on night data does not seem to improve nighttime detection performance and that such a model also works quite well at daytime. Next, we determined that training on correlated (rather than fully uncorrelated) images, i.e. taken from a smaller set of videos, does lead to a noticeably degraded model, however the difference is not as severe as we would have expected. Finally, we found that finetuning a model well trained at 640×640 resolution for a few additional epochs with 1280×1280 images leads to a considerate increase in detection performance when predicting at the higher resolution. Next, we compared our tracking method with several baselines and saw that a simple combination of a general purpose YOLOv3 object detector and a basic SORT tracker is insufficient for our problem at hand. However, replacing the detection network with our retrained and finetuned YOLOv5 model dramatically improves performance to a point where the system could be considered viable for practical applications. Further adding our SORT$^+$ enhancements leads to noticeable additional gains, especially with respect to the amount of objects detected and the consistency of maintaining tracks. At last, we carried out a performance study with accompanying analysis of our final end-to-end tracking tool. This demonstrated that our tracking system prototype already works well and comfortably runs in real-time. Further, we identified common failure cases that may be resolved in future work.

All in all, we successfully addressed all the research questions posed in Section 1.2 and

eventually managed to develop a self-contained tracking tool that works well in rural nighttime driving situations such as those most relevant for the CarVisionLight project, while at the same time also operating in real time. We hope that this tool, as well as some of our other insights, will prove useful in the CarVisionLight project and beyond. Therefore, we highlight several promising future research directions in the next section.

5.2 Future Work

In general, there is significant potential for future work based on the results of this thesis. We now discuss a few ideas that we consider particularly interesting and promising.

First, in this work we primarily trained object detection models on big publicly available datasets. This was mostly due to the fact that not enough labeled data of our particular problem domain, i.e. CVL videos, were available. We believe that revisiting our finetuning experiments with larger amounts of relevant labeled data has good potential to yield noticeable improvements with respect to the detection performance on the CVL evaluation data. Further, based on the observed output quality, we also think that our tracking tool could be very helpful in facilitating faster labeling of additional CVL videos. The initial tracking results of our tracker may be further refined by offline and/or single-object tracking methods before acting as a reasonable starting point for a human labeler that corrects sporadic mistakes and handles situations too complex for the tracker. We plan to develop such a semi-supervised labeling system in future.

Our tool already differentiated between different types of vehicles such as cars, trucks and buses. However, for an advanced driver assistance system, especially for an intelligent light controller, it would certainly also be useful to know the state of other vehicles, i.e. whether or not they are currently driving and, if so, in which direction (the same or the opposite). In most cases, this state is rather easy to determine based on the color (or absence) of the car's lights, hence a neural network model should easily be able to learn this given an appropriately labeled dataset. In fact, we guess that even a linear model (in form of a convolutional filter) operating on just the final feature map in the detection neural network may already yield decent results. Such an approach would probably only require a training dataset of modest size.

Another undoubtedly useful extension to our tracking tool would be computing distance estimates. As a start, one could utilize the fact that most cars have relatively similar sizes to calculate coarse, but in many situations still reasonable, distance approximations from just the detections of a single image coupled with the relevant camera parameters. If a stereo camera setup is available, more precise distances could be determined by applying our tracker to both views followed by a stereo matching step on the detections. Although this would of course halve the tracking speed, 15 FPS may still be acceptable in several applications. If not, one could try to decrease the prediction resolution of the underlying neural network model and/or trade off the accuracy of distance estimates for a higher frame rate by computing detections on both camera images only once every few frames.

Those were only some of the most immediate follow-up ideas. Thinking more long-term, they are many more interesting research directions, from extending our approach to 3D-tracking, over trying to also detect classes like lanes or the driveable area, to building an actual assistance system around our tracking tool, to name but a few. All in all, we are certain that much exciting research related to nighttime vision for driving applications will take place in the years to come.

List of Figures

List of Tables

Acronyms

AP Average Precision.

BDD Berkeley Deep Drive.

BDD-det Berkeley Deep Drive - Detection.

Bdd-track Berkeley Deep Drive - Tracking.

BLAS Basic Linear Algebra Subsystem.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

CVL Car Vision Light.

EMA Exponential Model Average.

FPS Frames Per Second.

GPU Graphics Processing Unit.

JSON JavaScript Object Notation.

mAP mean Average Precision.

MOT Multiple Object Tracking.

MOT17 Multiple Object Tracking Challenge 2017.

MS-COCO Microsoft Common Objects in Context.

NMS Non-Maximum Suppression.

SORT Simple Online Realtime Tracking.

SORT$^+$ Simple Online Realtime Tracking (Enhanced).

SSD Solid State Drive.

YOLO You Only Look Once.

Bibliography

[AAA+16] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[ABJ+08] Pablo F Alcantarilla, Luis M Bergasa, Pedro Jiménez, Miguel A Sotelo, Ignacio Parra, David Fernandez, and Silvia S Mayoral. Night time vehicle detection for driving assistance lightbeam controller. In *IEEE Intelligent Vehicles Symposium*, pages 291–296, 2008.

[ABJ+11] Pablo Fernández Alcantarilla, Luis Miguel Bergasa, Pedro Jiménez, Ignacio Parra, David Fernández Llorca, MA Sotelo, and Silvia S Mayoral. Automatic lightbeam controller for driver assistance. *Machine Vision and Applications*, 22(5):819–835, 2011.

[BGO+16] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple online and realtime tracking. In *IEEE International Conference on Image Processing (ICIP)*, pages 3464–3468, 2016.

[BMLT19] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixe. Tracking without bells and whistles. In *IEEE International Conference on Computer Vision*, pages 941–951, 2019.

[BS08] Keni Bernardin and Rainer Stiefelhagen. Evaluating multiple object tracking performance: the CLEAR MOT metrics. *EURASIP Journal on Image and Video Processing*, 2008:1–10, 2008.

[BWL20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. YOLOv4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[CBL+19] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A multimodal dataset for autonomous driving. *arXiv preprint arXiv:1903.11027*, 2019.

[Che09] Yen-Lin Chen. Nighttime vehicle light detection on a moving vehicle using image segmentation and analysis techniques. *WSEAS Transactions on Computers*, 8(3):506–515, 2009.

[CHL+19] Yuntao Chen, Chenxia Han, Yanghao Li, Zehao Huang, Yi Jiang, Naiyan Wang, and Zhaoxiang Zhang. SimpleDet: A simple and versatile distributed framework for object detection and instance recognition. *Journal of Machine Learning Research*, 20(156):1–8, 2019.

[Cho15] Wongun Choi. Near-online multi-target tracking with aggregated local flow descriptor. In *IEEE International Conference on Computer Vision*, pages 3029–3037, 2015.

[CV18] Zhaowei Cai and Nuno Vasconcelos. Cascade R-CNN: Delving into high quality object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 6154–6162, 2018.

[DVG18] Dengxin Dai and Luc Van Gool. Dark model adaptation: Semantic image segmentation from daytime to nighttime. In *IEEE International Conference on Intelligent Transportation Systems*, pages 3819–3824, 2018.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, Cambridge, MA, 2016.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[Gir15] Ross Girshick. Fast R-CNN. In *IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[GSG20] Florian Groh, Dominik Schörkhuber, and Margrit Gelautz. A tool for semi-automatic ground truth annotation of traffic videos. *Electronic Imaging*, pages 200–1 – 200–7, 2020.

[HDS+19] Martin Hahner, Dengxin Dai, Christos Sakaridis, Jan-Nico Zaech, and Luc Van Gool. Semantic understanding of foggy scenes with purely synthetic data. In *IEEE Intelligent Transportation Systems Conference*, pages 3675–3681, 2019.

[Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(9):1904–1916, 2015.

[JZL+19] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019.

[KB15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2015.

[KPG20] Shyamgopal Karthik, Ameya Prabhu, and Vineet Gandhi. Simple unsupervised multi-object tracking. *arXiv preprint arXiv:2006.02609*, 2020.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[LAE+16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016.

[LBK17] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems*, pages 700–708, 2017.

[LCWZ19] Yanghao Li, Yuntao Chen, Naiyan Wang, and Zhaoxiang Zhang. Scale-aware trident networks for object detection. In *IEEE International Conference on Computer Vision*, pages 6054–6063, 2019.

[LDG+17] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017.

[LGG+17] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision*, pages 2980–2988, 2017.

[LHB+08] Antonio López, Jörg Hilgenstock, Andreas Busse, Ramón Baldrich, Felipe Lumbreras, and Joan Serrat. Nighttime vehicle detection for intelligent headlight control. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 113–124, 2008.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision*, pages 740–755, 2014.

[LQQ+18] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8759–8768, 2018.

[MG15] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition*, 2015.

[MLTR+16] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. MOT16: A benchmark for multi-object tracking. *arXiv preprint arXiv:1603.00831*, 2016.

[OGJ08] Ronan O'Malley, Martin Glavin, and Edward Jones. Vehicle detection at night based on tail-light detection. In *International Symposium on Vehicular Computing Systems*, 2008.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[RF17] Joseph Redmon and Ali Farhadi. YOLO9000: Better, faster, stronger. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017.

[RF18] Joseph Redmon and Ali Farhadi. YOLOv3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[RHK17] Stephan R Richter, Zeeshan Hayder, and Vladlen Koltun. Playing for benchmarks. In *IEEE International Conference on Computer Vision*, pages 2213–2222, 2017.

[RTG+19] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 658–666, 2019.

[Sal14] Govind Salvi. An automated nighttime vehicle counting and detection system for traffic surveillance. In *IEEE International Conference on Computational Science and Computational Intelligence*, volume 1, pages 131–136, 2014.

[SDG19] Christos Sakaridis, Dengxin Dai, and Luc Van Gool. Guided curriculum model adaptation and uncertainty-aware evaluation for semantic nighttime image segmentation. In *IEEE International Conference on Computer Vision*, pages 7374–7383, 2019.

[SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. FaceNet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015.

[TL19] Mingxing Tan and Quoc V Le. EfficientNet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[TPL19] Mingxing Tan, Ruoming Pang, and Quoc V Le. EfficientDet: Scalable and efficient object detection. *arXiv preprint arXiv:1911.09070*, 2019.

[UVDSGS13] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[WBP17] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple online and realtime tracking with a deep association metric. In *IEEE International Conference on Image Processing*, pages 3645–3649, 2017.

[WZLW19] Zhongdao Wang, Liang Zheng, Yixuan Liu, and Shengjin Wang. Towards real-time multi-object tracking. *arXiv preprint arXiv:1909.12605*, 2019.

[YCW+20] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving dataset for heterogeneous multitask learning. In *Conference on Computer Vision and Pattern Recognition*, pages 2636–2645, 2020.

[ZDH+18] Lei Zhu, Zijun Deng, Xiaowei Hu, Chi-Wing Fu, Xuemiao Xu, Jing Qin, and Pheng-Ann Heng. Bidirectional feature pyramid network with recurrent attention residual modules for shadow detection. In *European Conference on Computer Vision (ECCV)*, pages 121–136, 2018.

[ZKK20] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. Tracking objects as points. *arXiv preprint arXiv:2004.01177*, 2020.

[ZPIE17] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *IEEE International Conference on Computer Vision*, pages 2223–2232, 2017.

[ZWK19] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.