Norwegian University of
Science and Technology

Autonomous Target Detection and Tracking for Remotely operated Weapon Stations

Vetle Bjørngaard Gundersen

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Jan Tommy Gravdahl, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This Master's thesis is a proposition of a novel system implementation for autonomous target detection and tracking, based on state-of-the-art computer vision methods. The assignment of exploring a relevant solution to combine a detector and a tracker was given by Kongsberg Defence & Aerospace (KDA), division Protech, and was carried out in the Spring semester of 2018. One of the main considerations for the tracking application is future deployment on the Nvidia Jetson TX2 module, which is an embedded platform. This is to make detection and tracking possible for a PROTECTOR Remote Weapon Station (RWS), a product delivered by KDA. The starting point for this thesis is my own preliminary project [1], from the Fall semester of 2017. This was a literature study on relevant computer vision methods for an autonomous object detection and tracking system, and the results from the preliminary work are the methods used in this thesis. For a complete presentation of the background and methods, some of the sections in this thesis are based on work presented in the preliminary project report.

Trondheim, June 9, 2018

Vetle Bjørngaard Gundersen

Acknowledgements

I am thankful for all the support, motivation and feedback from both friends and family during my work with this thesis. In addition to this, I would like to extend my sincere gratitude to ...

... **Professor Jan Tommy Gravdahl**, my supervisor at the Department of Engineering Cybernetics at NTNU, for supporting a thesis outside of his own area of expertise. His guidance has been important for continuous progress, and I believe that this thesis has been enlightening for the both of us.

... **Erik Marius Gamborg**, my supervisor and Senior Software Engineer at Kongsberg, for showing trust in my decisions and providing feedback and ideas at odd hours. I appreciate the opportunity to carry out a rewarding assignment for Kongsberg Defence & Aerospace.

... **Sara A. Kjærvik**, for pointing out the necessity of proofreading and selflessly taking her time to do so with parts of this thesis.

... **Sibel K. Solberg**, for being my rock. I would never be where I am today without her infinite love and support.

Abstract

This Master's thesis proposes a novel implementation of an autonomous tracker in Python, which combines a deep learning detection module and a point based tracking module. An accurate detection will introduce latency if the video capture rate exceeds the processing rate. The use of a frame buffer, a key element of the combination design, will compensate for this weakness. All frames periodically skipped by the detector will be stored, and a fast tracker will process the buffer to provide an updated object prediction for the current frame. The system implementation is developed with focus on future deployment on a Nvidia Jetson TX2 embedded platform, and utilizes Google's TensorFlow object detection API and the OpenCV object tracking API. The autonomous tracker is evaluated on a number of relevant videos, with a hybrid measure combining the bounding box overlap and a new proposed distance error score. The final system configuration, with a lightweight neural network for detection and the median flow algorithm for tracking, show real-time performance on a quad-core CPU.

Sammendrag

Denne masteroppgaven foreslår en ny implementasjon av en autonom målfølger i Python, som kombinerer en deteksjonsmodul basert på dyp læring og en punktbasert målfølgingsmodul. En nøyaktig deteksjon vil introdusere forsinkelse hvis frekvensen på henting av nye bilder fra videoen overstiger frekvensen av deteksjoner som leveres. Bruken av et bildebuffer, som er et hovedelement i kombinasjonsdesignet, vil kompensere for denne svakheten. Alle bildene som da jevnlig blir utelatt av detektoren vil bli lagret, og en rask målfølger vil gå igjennom bufferet for å levere et oppdatert objektforslag for det nåværende bildet. Systemimplementasjonen er utviklet med fokus på fremtidig distribusjon for en Nvidia Jetson TX2 innebygd [embedded] platform, og bruker Googles TensorFlow objektdeteksjons-API og OpenCV målfølgings-API. Den autonome målfølgeren vurderes mot en rekke relevante videoer, med et hybridmål som kombinerer overlappet til en avgrensningsboks [bounding box] og et nytt mål for avstandsfeil. Den endelige systemkonfigurasjonen, med et lettvekts nevralnett for deteksjon og «median flow»-algoritmen for målfølging, viser sanntidsytelse på en firekjærnet CPU.

Contents

Preface i

Acknowledgements iii

Abstract v

Sammendrag vii

Contents ix

List of Figures xiii

List of Tables xv

Acronyms xvii

1 Introduction 1
1.1 Background . . . 1
1.1.1 Motivation . . . 1
1.1.2 Remote Weapon Station . . . 2
1.1.3 Object Detection and Tracking Methods . . . 3
1.2 Problem Description . . . 6
1.3 Approach . . . 7
1.4 Contributions . . . 8
1.5 Related Work . . . 9
1.6 Structure of the Thesis . . . 10

I Theory 13

2 Computer Vision Fundamentals 15
- 2.1 Image Processing 17
  - 2.1.1 Filtering 17
  - 2.1.2 Feature Extraction 19
  - 2.1.3 Feature Description 23
- 2.2 Machine Learning 27
  - 2.2.1 Learning 27
  - 2.2.2 Models for Supervised learning 28

3 Computer Vision Application 33
- 3.1 Object Detection 35
  - 3.1.1 Definition 35
  - 3.1.2 Detection Method 35
  - 3.1.3 Pipeline 36
  - 3.1.4 Deep Learning 37
- 3.2 Object Tracking 41
  - 3.2.1 Definition 41
  - 3.2.2 Tracking Method 42
  - 3.2.3 Pipeline 42
  - 3.2.4 Point Tracking 45
- 3.3 Training and Evaluation 47
  - 3.3.1 Data Structuring 47
  - 3.3.2 Performance Measures 49
  - 3.3.3 Proposition of a Hybrid Tracking Measure 53

II Implementation 57

4 Software 59
- 4.1 Frameworks and Supportive Software 61
  - 4.1.1 Deep Learning Framework 61
  - 4.1.2 Tracker Implementations 66
  - 4.1.3 Security 66
- 4.2 Available Computer Vision Data 68
  - 4.2.1 Benchmark Data Sets 68
  - 4.2.2 Custom Test Videos 69
  - 4.2.3 Deep Learning Models 71

5 System Implementation **73**
5.1 Design . . . 75
5.1.1 Approach . . . 75
5.1.2 Limitations . . . 77
5.2 Implementation . . . 78
5.2.1 Modules . . . 78
5.2.2 Application Work-flow . . . 79
5.2.3 Post Processing of Raw Predictions . . . 81
5.2.4 Video Display . . . 83
5.2.5 Command Line Interface . . . 83
5.2.6 Data Processing Scripts . . . 87
5.3 Installation . . . 88
5.3.1 Host . . . 88
5.3.2 Target . . . 88

III Evaluation **89**

6 Results **91**
6.1 Tracking in Test Videos . . . 92
6.1.1 Video: HobbyKing . . . 92
6.1.2 Video: BlurBody . . . 93
6.1.3 Video: Dancer2 . . . 94
6.1.4 Video: David3 . . . 95
6.1.5 Video: Human2 . . . 96
6.1.6 Video: Jump . . . 97
6.1.7 Video: Woman . . . 98
6.1.8 Total Tracking Performance . . . 99

7 Discussion **101**
7.1 The Autonomous Tracker . . . 101
7.1.1 Proof of Concept . . . 101
7.1.2 The Embedded Platform . . . 102
7.1.3 Limitations . . . 102
7.2 Performance in Test Videos . . . 104
7.2.1 The Selected Videos . . . 104
7.2.2 Evaluation of the Results . . . 104
7.2.3 Bounding Box Consideration . . . 106
7.2.4 The Proposed Hybrid Measure . . . 107

8 Conclusion and Further Work **109**
8.1 Conclusion . . . 109
8.2 Further Work . . . 110
8.2.1 Work-flow . . . 110
8.2.2 Implementation . . . 110
8.2.3 Data Set . . . 111

A The GitHub Repository **113**
A.1 detection-and-tracking/ . . . 114
A.2 host/ . . . 116
A.3 host/scripts/ . . . 118
A.4 host/test/ . . . 120
A.5 target/ . . . 122
A.6 videos/ . . . 125

Bibliography **127**

List of Figures

1.1 PROTECTOR RWS . . . 2
1.2 Computer vision structure . . . 4
1.3 Intuitive system model . . . 7

2.1 Image representation . . . 17
2.2 Edge detection . . . 20
2.3 Harris corner response . . . 22
2.4 Corner detection . . . 22
2.5 Difference of Gaussian . . . 24
2.6 SIFT descriptor steps . . . 24
2.7 Example of HOG features . . . 26
2.8 SVM - Kernel trick . . . 29
2.9 Illustration of a neuron . . . 30
2.10 Simple mathematical model of a neuron . . . 30
2.11 Activation functions . . . 31
2.12 Typical structure of an Artificial Neural Network . . . 32

3.1 Detection pipeline . . . 36
3.2 Structure of CNN . . . 38
3.3 Faster R-CNN . . . 40
3.4 Tracking pipeline . . . 42
3.5 Extended tracking pipeline . . . 43
3.6 Limitations of point tracking . . . 45
3.7 State variable update . . . 46
3.8 Overfitting . . . 48
3.9 Data set separation . . . 49

3.10 Center distance error 54
3.11 Distance Score Illustration 55

4.1 TensorRT work-flow 63
4.2 Custom videos 70

5.1 Frame processing timeline 76
5.2 System decision tree and module interaction 76
5.3 Video overlay 84

6.1 Frames from tracking in HobbyKing.mp4 92
6.2 Performance plot for HobbyKing.mp4 92
6.3 Frames from tracking in BlurBody.mp4 93
6.4 Performance plot for BlurBody.mp4 93
6.5 Frames from tracking in Dancer2.mp4 94
6.6 Performance plot for Dancer2.mp4 94
6.7 Frames from tracking in David3.mp4 95
6.8 Performance plot for David3.mp4 95
6.9 Frames from tracking in Human2.mp4 96
6.10 Performance plot for Human2.mp4 96
6.11 Frames from tracking in Jump.mp4 97
6.12 Performance plot for Jump.mp4 97
6.13 Frames from tracking in Woman.mp4 98
6.14 Performance plot for Woman.mp4 98
6.15 Average tracking scores for the test videos 99

7.1 Difference in bounding box approach 106

List of Tables

1.1 Comparison of object detection approaches 5
1.2 Comparison of object tracking methods 5

3.1 Prediction outcomes for object detection 50
3.2 Calculation of mAP 51

4.1 Deep learning frameworks 62
4.2 License permissions 65
4.3 License conditions 65
4.4 License limitations 65
4.5 Custom test videos 70
4.6 Object detection models 71

6.1 Performance summary 99

Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
CNN	Convolutional Neural Network
CV	Computer Vision
DoG	Difference of Gaussian
FOV	Field of View
FPS	frames per second
GPU	Graphics Processing Unit
HOG	Histogram of Oriented Gradients
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoU	Intersection over Union
LoG	Laplacian of Gaussian
mAP	mean average precision

R-CNN Region based CNN
RoI Region of Interest
RPN Region Proposal Network
RWS Remote Weapon Station

SIFT Scale Invariant Feature Transform
SVM Support Vector Machine

Chapter 1

Introduction

1.1 Background

This section is based on sections 1.1 and 1.2 of the preliminary project [1].

1.1.1 Motivation

The research area of Computer Vision (CV) can be traced back to the late 1960's, and was meant to mimic the human vision system for creation of robots with intelligent behaviour. The "visual input" problem was at this time believed to be an easy step along the way of more complex systems. A student at MIT was simply assigned a task to "spend the summer linking a camera to a computer and getting the computer to describe what it saw" [2, p. 781]. Today, this statement can be seen as a great underestimation of the problem of having computers acquire, process, analyze, and understand digital images the way humans do.

Digitalization of visual input is a form of innovation that is and will be the future of many industries. As with all innovation, the field of CV is driven by technological advancements, industrial competition and an ongoing need for better solutions. First, the development of the Graphics Processing Unit (GPU), using dedicated hardware to process images, made visual solutions more attractive. Then, competition pushes the need for better visual sensors, as with the automotive industry and their development of driver-less cars. Furthermore, the application area of human assistance emerges as an important factor. This can for instance be interpretation of medical images, or detection of abnormal behaviour in surveillance videos.

Finally, there is a demand for new solutions to counter the issues that arise in the wake of new technology. Consumer-friendly drones raise important issues on how abuse of technology may threaten privacy and security. This was uncovered when a drone landed on the lawn of the White House in 2015 - undetected [3]. And more recently, 2018 became the year of “the first announced use of a swarm of drones in a military action” [4]. From a security and defence perspective, the application of CV for object detection and tracking is more relevant than ever.

1.1.2 Remote Weapon Station

A Remote Weapon Station (RWS), as seen in Figure 1.1, is a platform for light and medium range weapons, mounted on top of a vehicle or any other surface [5]. This is a system which enables an operator to handle weapons, as well as observe targets through different cameras, in a more protected environment; seated inside a vehicle or at another remote location. To further develop this remote system a natural step would be, if not automate target acquisition completely, to assist an operator in the process of finding targets. For the case of the PROTECTOR RWS, this will include taking advantage of the camera video already available, which is displayed on a control screen for the operator.

Figure 1.1: Image of a PROTECTOR Remote Weapon Station (RWS).

The application platform for an autonomous target detector and tracker should be irrelevant. Regardless of it being an autonomous vehicle, a RWS mounted on a vehicle or a stationary camp protection, the task at hand will remain the same; find the target and keep hold of it. For the field of computer vision this can be translated into detection and tracking of objects.

1.1.3 Object Detection and Tracking Methods

In the preliminary project [1], the objects *drone*, *person* and *vehicle* were discussed as relevant targets for the application area of the RWS. These objects are relevant target classes from a defence perspective, but they are also trending objects in other CV research and application areas. The most important issue to address for these objects, is the representation difficulty. When taking their physical attributes into account, it is difficult to find common ground of which to simplify a representation model. For their structure, both drones and vehicles are rigid objects, but a person is non-rigid. The motion pattern for both person and vehicle tend to be somewhat predictable, but drones can fly in irregular and unpredictable patterns. In size these three objects are all usually different, ranging from small to large.

Several surveys on detection and tracking were used to organize the most popular methods and algorithms [1]. The resulting structure is illustrated in Figure 1.2. Furthermore, relevant methods in light of the RWS application environment were evaluated by qualitative measures. A detector can be compared on attributes such as decision accuracy, processing speed and generality. A tracker can be compared on its handling of occlusion, entry/exit of objects, whether it can be optimized and if it can handle more than one object at the same time. Table 1.1 and Table 1.2 show a summary of the findings. Each method category is reduced to only show the qualities of the method with the best attributes. From these tables, it is suggested that an accurate *deep learning* detector could be combined with a fast *point based* tracking method, for an acceptable result in the RWS application area.

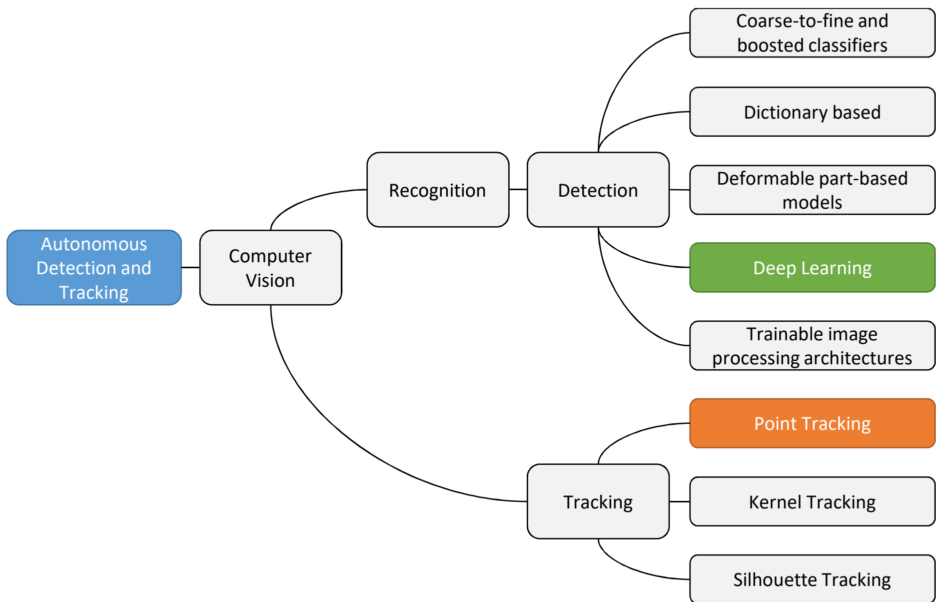


Figure 1.2: Sub-categories of computer vision, when used for the task of autonomous detection and tracking. Both detection and tracking can be divided into several methods.

Table 1.1: Qualitative comparison of object detection approaches, based on Table 1 in [6], and originally presented for the preliminary project [1].

Category	Accuracy	Speed	Generality
Coarse-to-fine and boosted classifiers	Good	Good	Bad
Dictionary based	Medium	Medium	Good
Deformable part-based models	Good	Bad	Medium
Deep learning	Good	Medium	Good
Trainable image processing architectures	Medium	Medium	Medium

Legend: Good, Medium, Bad

Table 1.2: Qualitative comparison of object tracking methods, based on table in [7], and originally presented for the preliminary project [1]. The table indicates the events of multiple objects, object entry, exit and occlusion, and if the method provides an optimal solution to the cost function minimized.

Category	Occlusion	Entry	Exit	Optimal	Multiple
Point tracking	√	√	√	√	√
Kernel tracking	√	X	X	X	√
Silhouette tracking	√	X	X	√	√

Legend: √ Yes, X No

1.2 Problem Description

This thesis will focus on autonomous functionality for remotely operated weapon stations. A RWS typically provides one or more cameras in addition to a control interface for an operator. State-of-the-art CV methods can therefore be used on the camera video to automate certain tasks, which can be useful in several scenarios. One example is camp protection, in which there may be more than one weapon station per operator. In this case it will be useful with a system to assist the operator in finding and surveilling targets. The weapon station will provide the ability to search for relevant targets, and track them until the operator can evaluate them and decide on the appropriate course of action. This kind of autonomous functionality can also be useful for a weapon station mounted on an autonomous vehicle. The weapon station can then autonomously look for targets while the vehicle is on the move, and alert the operator when there are potential threats. Both scenarios can benefit from an accurate target detector back-end for finding potential targets. A target tracker front-end can then maintain track of the target until the operator has time to consider it.

With the background as described in section 1.1, and the preliminary literature study [1] of applicable CV methods, this thesis will describe and test a novel implementation of a detection module combined with a tracking module. The motivation for combining two such modules explicitly, is because of the real-time demands on a vision based system. The hazardous working environment of a RWS requires the system to process live camera video, and present instant feedback to the operator - that is, track target objects in real-time. However, a thorough processing of images takes time, in which there is an obvious trade-off in accuracy and speed for a detector. A solution to this compromise will be explored in terms of designing a system to initialize a fast tracker with accurate detections - resulting in an autonomous object tracker.

The tracking application will be implemented with focus on running on an embedded platform in the future, specifically the Nvidia Jetson Tegra X2 development module (Jetson TX2). This is to keep the tracking system close to the RWS, to minimize video transportation overhead, and create a general module to be installed for different scenarios. With rights to this hardware, a relevant software framework must be chosen to realize deep learning for object detection and a point based tracking algorithm.

1.3 Approach

The problem of providing autonomous assistance to an operator of a weapon station, as in the scenarios described in section 1.2, is put together by a series of complex issues. From choosing the right video input format and processing the video, to providing final results in form of graphics - or even regulation of weapon station movements. This thesis will therefore only focus on the video processing, and present performance results in a convenient manner for discussion purposes.

Problem steps
With this thesis, the problem will be broken down into the following steps:

1. Introduce CV operations and the suitable detection and tracking methods as determined by the preliminary work [1].
2. Choose a relevant software framework to implement object detection and tracking.
3. Research relevant data sets of images for the deep learning approach, as well as videos for performance testing of the final tracking application.
4. Create a module based system architecture for the combination of detection and tracking, with focus on a solid foundation for further development. The idea of how a tracker can be periodically corrected by a detector is illustrated in Figure 1.3.
5. Explore implementation of a *single target* tracking system for the Jetson TX2 embedded platform.

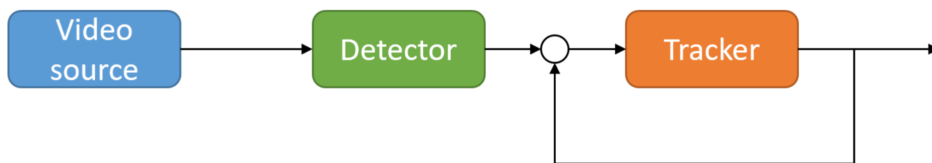


Figure 1.3: An intuitive system model, combining a detector and a tracker in a feedback loop. The tracker is initialized by the detector, with a periodic correction.

Assumptions
This approach to the problem is based on the following assumptions:

1. The research area of CV is not lectured by the department of Engineering Cybernetics, so an introduction to basic image processing is necessary.
2. The object class *drone* is the most relevant class in terms of recent defence activities, but for tracker evaluation it is the drone *characteristics* that are interesting. Any similar class of small objects with irregular motion patterns can compensate for lack of annotated drone data, and can replace it for testing.
3. Future work on object tracking for embedded deployment will benefit from a solid software foundation, and a "best practice" use of the third-party tools utilized in the system.
4. The system design and architecture developed in this thesis is not in any way claimed to be optimal. The system is provided as a proof of concept based on an intuitive approach, but nevertheless implemented with focus on maintainability for any further development.
5. The real-time constraint to consider is the definition of *live* camera video. For this problem, the constraint is measured in frames per second (FPS) - that is, the capture rate of the camera. For both 4:3 and 16:9 frame aspect ratios, 30 FPS is a commonly used and reasonable capture/display rate [8].

1.4 Contributions

This thesis provides the following contributions:

1. Practical use of state-of-the-art CV methods, with focus on future RWS applications
2. Proposition of a new center distance score, and a hybrid measure, for tracker performance
3. A solution to how a detection module and a tracking module can be combined in practice
4. Implementation of an autonomous tracking application with real-time performance
5. Open source GitHub repository [9] (see Appendix A) for further development

1.5 Related Work

A consideration of related work for the objective of the thesis may give a brief insight of what to expect. When regarding the embedded platform Jetson TX2, a proposal of a computer vision system for detection and tracking is previously provided for an older Jetson platform [10]. This implementation for a Jetson *TK1* (an older model) utilizes software tools customized for the platform, in addition to the deep learning framework *Caffe*, to implement the use of a convolutional neural network (CNN). The authors also stress how a combination of CV modules demand high-performance embedded-processing hardware, such as a GPU, to enable real-time behaviour. This is supported by another paper with a CNN based tracking implementation [11], which states that a better GPU is required to train and test a neural network in reasonable time. The implementation proposed show how the classification ability of a CNN can be used in a tracking task.

With an interest in drones, a large data set with representative images of this class will be required for a deep learning approach - which is assumed to be an issue. One solution to the scarce data problem is to create artificial images. A very recent paper [12] describes an algorithm for posting images of drones at random positions in frames taken from a video. With control of the object placement, associated ground truth annotations are created as well. The artificial data set is used for training of a CNN detection model, and proved successful by detecting drones and distinguishing them from birds.

These papers are related to the problem of this thesis in terms of platform focus, the deep learning approach and data set consideration. This serves as an indication of what to expect in the process of implementing a computer vision based application. With regards to the Jetson TX2, it will be important to focus the development on software compatible with the limited hardware of the platform. A future deployment will benefit from the use of versatile development tools and software with support of runtime optimization. In case of scarce data for a specific object class, the system can still be evaluated with another class to illustrate application behaviour - with the notion that a data set can be artificially generated at a later step in the development process.

1.6 Structure of the Thesis

The structure of the thesis is designed to reflect the sub-tasks presented in section 1.3. An overall chapter sectioning, in three parts, is enforced to illustrate the development process of the tracking application.

Part I - Theory
The first part serves as an introduction to CV, and the relevant methods for the problem approach.

> **Chapter 2**: Introduces basic image processing operations, which is the foundation of CV. From image filtering and feature description, to different architectures for machine learning.
>
> **Chapter 3**: Provides definitions on visual object detection and tracking, introduce the methods intended for the implemented application, and the most common performance metrics for detection and tracking. In addition to this, the final subsection defines a new hybrid tracking measure to evaluate tracker performance.

Part II - Implementation
The second part is an elaboration on the software components of the system, with a complete description of the developed application.

> **Chapter 4**: Is a discussion of available software frameworks and data relevant to the detection and tracking application. It also highlights different aspects for consideration when relying on open source software.
>
> **Chapter 5**: Describes the system design and architecture for a command line application implemented in Python. This includes presentation of the intended work-flow, limitations and the final application interface.

Part III - Deployment
The third part is a discussion on the performance of the implemented application.

> **Chapter 6**: Presents the performance results of the implemented tracking application. It is important to notice that different data sets and videos are used to illustrate the different aspects of this specific combination of detection and tracking methods.
>
> **Chapter 7**: Discuss the final tracking application and the performance results in terms of the objective of this thesis.
>
> **Chapter 8**: Presents concluding remarks on the proposed tracking application, and provides relevant notes on how to further develop the tracking system as it is described in this thesis.

Appendix A: Provides an overview of the publicly available, open source, GitHub repository created for the purpose of this thesis.

Part I

Theory

“Simplification is one of the most difficult things to do.”

Sir Jonathan Ive

Chapter 2

Computer Vision Fundamentals

Contents:
This chapter introduces basic image processing operations, which is the foundation of CV. From image filtering and feature description, to different architectures for machine learning.

Literature review:
In *Computer Vision: A Modern Approach* [13], Forsyth and Ponce present CV with a bottom-up approach; from a physical understanding of the image formation, to the high level application areas of object classification. CV as a field can be considered disorganized, and this is (from their preface) because it is an "intellectual frontier". The book succeeds in presenting an orderly picture of CV, based on decades of experience within the field. This is why it provides a good baseline for the theory of this report.

Even though the book [13] includes a broad range of application as an improvement in the second edition, the book *Computer Vision: Algorithms and Applications* [14] has this in focus. Szeliski provides a book with a more hands-on approach to CV, with algorithms that have real-world applications and work well in practice. This is based on his experience in computer vision research in corporate research labs. *Digital Image Processing* [15] is also used as a source to further cover the ground on general background mathematics, which is preferred before studying CV techniques. Basic image processing techniques are covered elaborately in this book, which also includes object recognition in the last chap-

ter. Finally, an introduction to artificial intelligence from *Artificial Intelligence: A Modern Approach* [16] is included in the source material. This is because learning is an important part of the application of CV, and the topic is treated in this book. Both the section on machine learning and artificial neural networks are relevant.

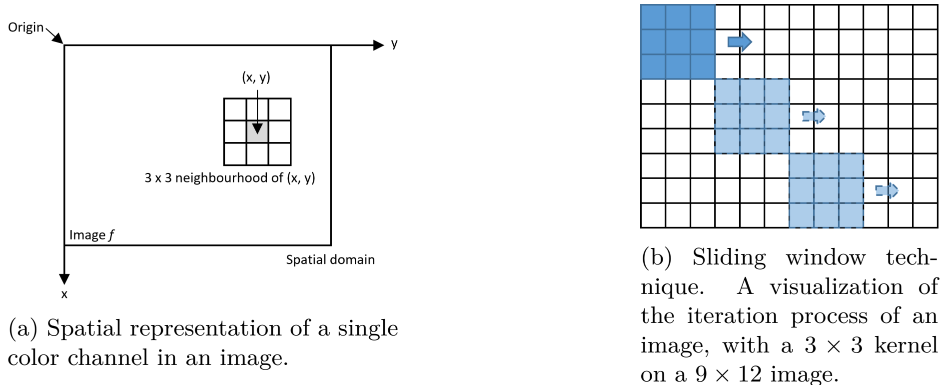
All of the books mentioned above are recommended and used as curriculum in image processing, computer vision and artificial intelligence courses at NTNU. They are all published after 2010, and most of them contain references to recent work done within the field of CV.

2.1 Image Processing

2.1.1 Filtering

This subsection is based on section 2.1 of the preliminary project [1].

With an image sensor, e.g. a camera, a scene is quantified into an n-dimensional map of discrete values. A basic grey image is a 2D-map of intensity values representing shades from black to white, and a Red-Green-Blue (RGB) color image is a 3D-map because of its multiple color channels defining each pixel. This type of color image, with channels/layers of red, green and blue combined to create different colors, can be defined as a three-dimensional intensity function $f(x, y, z)$. The intensity of color channel z is given at the spatial coordinates (x, y). With the origin $(0, 0)$ of a color channel usually defined as the top left corner, the spatial representation can be illustrated as in Figure 2.1a.



(a) Spatial representation of a single color channel in an image.

(b) Sliding window technique. A visualization of the iteration process of an image, with a 3×3 kernel on a 9×12 image.

Figure 2.1: Image representation.

Image processing, like sharpening or blurring, can from this kind of image representation be defined as mathematical operations. A new manipulated image can be created by combining a spatial filter (also called matrix/kernel/mask/template/window) with an image, a process called *filtering*. When combining a spatial filter with an image, the iteration over an $M \times N$ image can be visualized as a *sliding window* across the M rows and N columns, as it is illustrated in Figure 2.1b. The nature of the filtering process, i.e. the resulting image, is determined by the kernel used. A kernel with e.g. slightly negative values and a positive value in the middle would enhance contrasts and suppress smaller changes, which is a sharpening of the image.

On the other hand, a smoothing/blurring of the image would require the intensity values in a neighbourhood to be more equal to each other. An averaging filter, with a kernel to add intensity values in a neighbourhood and divide them by the kernel size, is a linear filter which creates this blurring effect. This is also the case for a Gaussian smoothing filter: a square kernel with values to replicate a Gaussian normal distribution, divided by the kernel sum. The Gaussian smoothing operation is basically a weighted average of a neighbourhood, with the most central pixel value contributing the most.

These filters, however different their kernels might be, are all kernels correlated with an image. This correlation can also be described as a *convolution* if the same kernel is rotated 180 degrees (flipping both rows and columns of the kernel). From this, a filtered image $g(x)$ can be defined as a convolution:

$$g(x) = w(x,y) * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t) \tag{2.1}$$

Equation 2.1 is the *discrete* convolution of a kernel w with a spatial image f. The image function f is a discrete function, because of the discrete intensity values at each position (x, y). (The convolution of a continuous function is originally described by an integral, which becomes a sum when the function is discrete). The reason why it is important to express a filtering process as a convolution, is because of the identity found in the convolution theorem. One half of the theorem is denoted in Equation 2.2:

$$f(t) * w(t) \iff W(\mu)F(\mu) \tag{2.2}$$

Equation 2.2 indicates that a convolution between two functions in the spatial domain, is a multiplication in the frequency domain. As long as the filtering operation is linear, it does not matter whether it is done in the spatial or frequency domain. In the spatial domain a filtering process is more intuitive, as of how the image is traversed with a kernel (sliding window), and how a neighbourhood is weighted. As for the frequency domain it is no point in visualizing the process, but the operation is easier to implement, and it is arguably faster to perform two discrete Fourier transforms and a matrix multiplication (of which the whole image is processed at once).

The key transformation for this implementation is the 2D discrete Fourier transform (DFT) [15, p. 257]:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{2.3}$$

Equation 2.3 denotes the DFT of an $M \times N$ digital image. To be able to use the identity given by the convolution theorem, the discrete Fourier transform and the inverse Fourier transform is used. The ability to do a transformation to the frequency domain and back is indicated with the double sided arrow in Equation 2.2.

2.1.2 Feature Extraction

This subsection is based on section 2.2 of the preliminary project [1].

The first kind of features to stand out when looking at an image are those with distinct shapes and boundaries. To be able to describe these we are interested in *edge segments* to create regions, and *corners* to be able to determine specific locations. The concept of these *key point features* or *interest points* will be further explained, with examples on how to extract them.

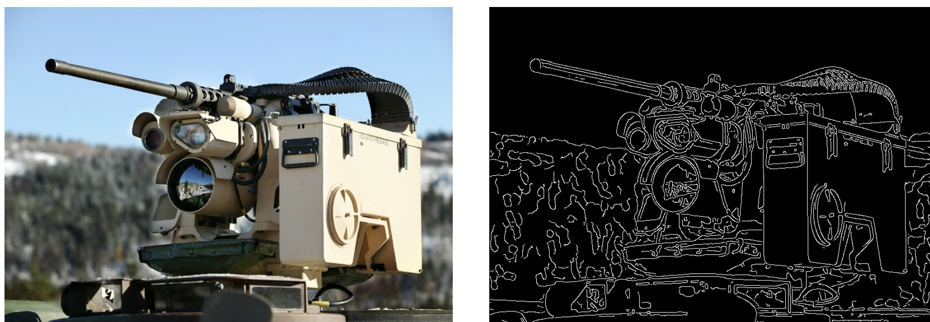
Edge
A first step towards any method to interpret an image is to extract information which describes the scene. Usually there is a distinction between relevant and irrelevant scene elements, which can be noted as the foreground and the background. To distinguish an element from its background intuitively we know that there are regions, even visual boundaries, that separate these. A boundary of a finite region creates a closed path, and can be seen as a "global" concept. To be able to determine a boundary, we use edges, which per definition is not necessarily a complete line: "The idea of an edge is a "local" concept that is based on a measure of intensity-level discontinuity at a point."[15, p.92] This means that when multiple edge points are linked together in segments, these edge segments can be linked to correspond a boundary, but it is not always the case. An edge segment, or just edge, is a curve that follows a path of rapid change in intensity.

Whether or not an edge segment (or just edge) can be defined as a boundary, edges are features which describes the content of an image. A common algorithm for extracting edges is the *Canny Edge detector*.

The steps of the algorithm [15, p.745]:

1. Smooth the input image with a Gaussian filter.
2. Compute the gradient magnitude and angle images.
3. Apply non-maxima suppression to the gradient magnitude image.
4. Use double thresholding and connectivity analysis to detect and link edges.

Step (1) is to prevent edges to be extracted from noise. The gradient is computed in step (2) because an edge is defined as an intensity discontinuity, and this is done with a derivative filter. With step (3) the algorithm provides a single edge point response, to only identify a single edge point with each true edge point. Finally, a double threshold is provided in step (4) to only preserve the strongest edge points (high threshold) and the weak edge points (low threshold) connected to a strong edge point. The output is a feature map of edge segments, as illustrated in Figure 2.2.



(a) Original image.

(b) Canny edge detection.

Figure 2.2: Edge detection of image.

Corner

Even though edges describe features in an image, it is difficult to determine which edge points are the same in two closely related images. When a window is translated along an edge in an image, it is not possible to determine a location because the image patch in the window will not change significantly. This is the same as sliding the window across a region of constant value: it does not provide any useful information (also known as the *aperture problem*). Even if it is to relate two camera models of a scene, or to determine the trajectory of an instance in sequential images, we are interested in features we can *localize*. A corner is a point with this attribute. It does not only provide a distinct location of discontinuity

in the intensity (edge), but a change in direction of an edge segment. A corner point can be identified with a specific (x, y)-value in an image f.

Intuitively, an idea would be to walk along an edge until it changes direction. The problem with this is that most edge detectors fail at corners, because corners are covered by a smoothing filter. For corners, there is a need for a more tailored algorithm to avoid this problem. A common algorithm for this is the *Harris corner detector*, with the following steps:

1. Spatial derivative calculation
2. Structure tensor setup
3. Harris response calculation
4. Non-maximum suppression

In step (1), the rate of intensity change is calculated. The classic Harris detector uses a simple 1D derivative filter, but in more resent years the image is rather convolved with the horizontal and vertical derivatives of a Gaussian. From the spatial derivatives, we can construct a structuring tensor S in step (2). The notation for this is different in both [13, p. 150] and [14, p. 212], but can be defined as S in Equation 2.4:

$$S(x, y) = \sum_{window} \{(\nabla I)(\nabla I)^T\} \approx \sum_m \sum_n w(m, n) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{2.4}$$

The $m \times n$ window entails the directional gradients, modified with a weight w which diminishes the contribution of positions that are far away from the central pixel (x, y). We are interested in the eigen values of this tensor, λ_1 and λ_2, to calculate the Harris response R in step (3). A simple quantity to use is defined in Equation 2.5:

$$R = \det(S) - \alpha \mathrm{trace}(S)^2 = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2 \tag{2.5}$$

With α being an empirically determined constant ($\alpha \in [0.04, 0.06]$). How the Harris response R is interpreted is illustrated in Figure 2.3. From step (4), only the pixels with a response above a given threshold is kept as a corner. An example of the result of a Harris corner detection is shown in Figure 2.4.

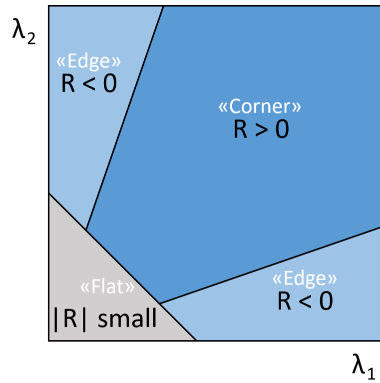


Figure 2.3: Illustration of Harris corner response: Relation between the magnitude of the structuring tensor eigenvalues (λ_1, λ_2), and the Harris response R.

Figure 2.4: Result of “Harris corner detection”.

2.1.3 Feature Description

This subsection is based on section 2.3 of the preliminary project [1].

After extracting interest points, the next step is to utilize them to describe the scene in the image. A *descriptor* is a way of constructing features from interest points. These features can be seen as more or less distinct patterns. Both SIFT (Scale Invariant Feature Transform) and HOG (Histogram of Oriented Gradients) are common methods to represent and describe the features.

SIFT: Scale Invariant Feature Transform
A Scale Invariant Feature Transform (SIFT) descriptor uses both magnitude and orientation of image gradients. The basic idea is to put image gradients into histograms, based on orientation and magnitude, but this will confuse similar patches of an image with each other. There are a few more steps to it, in order to create a more distinct descriptor. The steps of the SIFT algorithm can be summarized into the following (based on [13, p. 157] and [14, p. 223]):

1. Extract candidate/interest points from Difference of Gaussian (DoG) images.
2. Update location of candidate points by interpolating color values in neighbourhood.
3. Eliminate low contrast candidates and candidate points along the image edge.
4. Assign orientation to interest points, based on peaks in histogram of gradient directions in a small neighbourhood.
5. Histogram is normalized, thresholded and normalized again.

The first steps (1)-(3) are meant to extract distinct feature points, and the last two steps (4) and (5) are the construction of the descriptor. Step (1) is based on the concept that the DoG is an approximation of the Laplacian of Gaussian (LoG), and it is easier to compute. The LoG is a combined two-step of extracting image gradients, filtering the image with a *Laplacian computed Gaussian kernel*. The Gaussian filters out noise, and a Laplacian filter extracts image gradients. Multiple convolutions are then performed on an image, with different scaled Gaussian, which gives a stack of Gaussian images. This is illustrated in Figure 2.5. The difference of Gaussian is computed between two and two images, creating a DoG image stack. With step (2), the candidates are then extremes across scale images, which is an interpolation of the 3D neighbourhood with scale image above and below. Step (3) only keeps the stronger interest points.

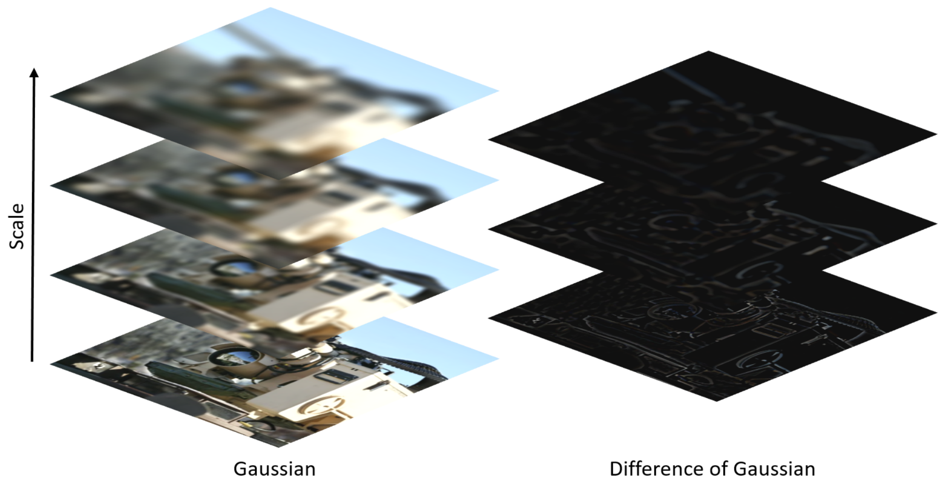


Figure 2.5: Difference of Gaussian. Image filtered by Gaussian with different scales, to approximate Laplacian of Gaussian.

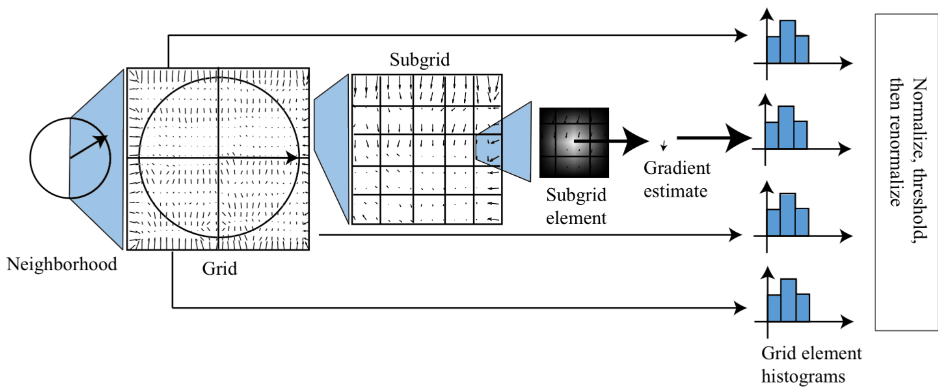


Figure 2.6: SIFT descriptor steps. (Source: [13, p. 157]).

The final steps are the interesting part of the SIFT construction, and are illustrated in Figure 2.6. The image is divided into a $n \times n$ grid, and each grid element into a $m \times m$ sub-grid. All pixels in a sub-grid element are represented in a histogram separated in q different orientations. Each gradients impact on the histogram is determined by magnitude, as well as its closeness to the center of the sub-grid. For the detector to be invariant to effects of change in illumination intensity, the histogram is normalized. Then, the histogram is cut by a threshold t, to reduce very large gradients which tend to be unstable. After this it is renormalized. The dominant orientation is decided by the peak in the histogram. A SIFT descriptor is a set of normalized histograms of image gradients.

HOG: Histogram of Oriented Gradients

The Histogram of Oriented Gradients (HOG) descriptor can be seen as a special case of the SIFT descriptor. Just as with SIFT, gradient orientations are organized in histograms, but for HOG features the process is adjusted to extract high-contrast edges. From the SIFT algorithm, the histogram is normalized over a neighbourhood, but for the HOG descriptor it is normalized with respect to nearby gradients only. Even though an image is divided into a grid with subgrids, the normalization can occur over a different sub-grid element other than the orientation sub-grid (initial division). With this overlapping normalization, a gradient can contribute to multiple histograms, with different weights and outcome. From [13, p. 160], we can express the weight of a gradient at position x for a cell $\mathcal{C}$ as in Equation 2.6:

$$w_{x,\mathcal{C}} = \frac{\parallel \nabla I_x \parallel}{\sum_{u \in \mathcal{C}} \parallel \nabla I_u \parallel} \tag{2.6}$$

The gradient magnitude $\parallel \nabla I_x \parallel$ is compared to the others in the cell. With this, the HOG features are good at extracting outline curves from confusing backgrounds, and is proven to be a good approach for human detection [17] (because of the distinct "lollipop" shape from a persons silhouette).

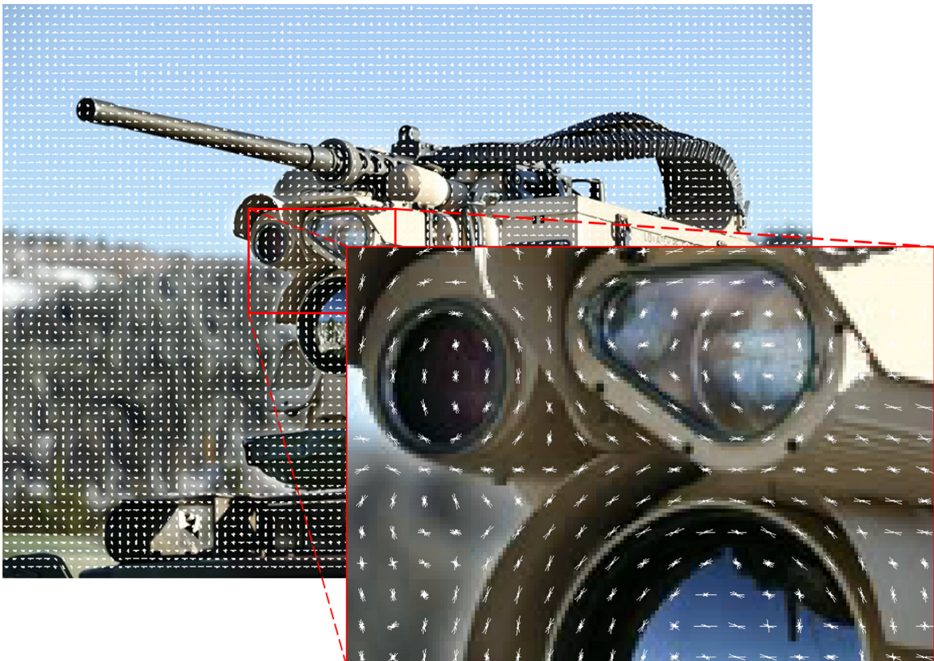


Figure 2.7: Example of HOG features, represented as rose plots.

2.2 Machine Learning

This section is based on section 2.4 of the preliminary project [1].

2.2.1 Learning

Learning, within the field of artificial intelligence, can be defined with the following statement:

> " An agent [something that acts] is *learning* if it improves its performance on future tasks after making observations about the world. "
> [16, p. 693]

The reason that a learning agent is attractive, instead of a designer implementing all improvements of a system at first, is because of at least one of the three following reasons [16]:

1. A designer can not anticipate all possible situations the agent will act in
2. A designer can not anticipate all changes over time
3. The designer does not know how to implement the solution

For CV, learning is interesting and useful for the purpose of *classification*. A feature descriptor is part of the process, of which the features in a new image should be matched to output a result. When the result is discrete we denote the output with a class *label* (and when the result is a continuous value, it is a regression problem). The general process of a classifier is to generate a weighted output based on a set of input values. This means that the core of a classifier lies in the structure of its weights: how they are combined and how they are determined and updated. We are interested in updating the weights by previous results, to improve the performance over time - thus creating the learning effect.

The learning process can be categorized by the *type of feedback*, and this gives three main types of learning [16, p. 694]: unsupervised, reinforced and supervised. Each type with gradually more informative feedback. *Unsupervised learning*, as the name implies, is not provided with any explicit feedback. The agent learns patterns from the input data, but it is only able to determine a structure or a way of sectioning this data. With unsupervised learning the goal is to create potentially useful groups of data from input examples, which is also known as *clustering*. *Reinforcement learning* is a more informed method of learning. The agent is not provided with any form of solution to the desired result from the input example, but it is provided with a set of possible rewards and punishments - *reinforcements*. With a given input example, the agent is given either a positive

or a negative feedback (i.e. a score). Based on this more or less basic feedback, the agent learns to associate the outcome from an input with something positive or negative, and will improve its actions to obtain a reward. Finally, the *supervised learning* method provides the most complete set of feedback. That is, the agent is provided with the solution (or a *ground truth*) for the problem it is set to solve. By observing the input-output pairs, it can map the required response to a function, and update its performance.

With focus on the RWS application environment, the CV learning process will consist in identifying targets and their location from a series of images. If this were to be specified as an unsupervised learning environment, it would be the same as saying that the targets are unknown, but would like to see if something in the environment stands out. As this is not the case, and there is a set of objects of interest (targets), all available information should be utilized. A supervised learning approach will require data sets with corresponding solutions, that is, images with content description to supervise the process. It is a better idea to provide the agent with a complete solution, if available, and optimize performance with rights to a labeled data set. There are several useful and popular models for supervised learning, with some further explained in subsection 2.2.2, but the Artificial Neural Network (ANN) can be seen as most relevant for the approach of this thesis.

2.2.2 Models for Supervised learning

Adaptive Boosting
The method of Adaptive Boosting, or *AdaBoost*, is one of the best off-the-shelf classification algorithms [18]. This is because the algorithm is put together by multiple "weak" classifiers, and the sum of their prediction gives a *boosted* classification. In [13, p. 475] the general boosting process is described as a series of classifiers trained to correct each others mistakes. The first classifier is trained and weighted on a data set, and the second classifier is trained and weighted to get examples right if the previous got them wrong. This goes on for a number of times, and the final classification is a weighted combination of the outputs of these classifiers. The resulting classifier is extremely fast in practice, but the training time can be very long (in order of weeks) [14, p. 664].

Support Vector Machine
A Support Vector Machine (SVM) is a statistical method for determining a hyperplane to separate two classes, in order to make binary decisions. With a data set of two classes, the SVM searches for a plane with maximum margin to the classes (that is, biggest gap between the closest samples) [14, p. 662]. With the

vectors of these closest samples, we can describe a hyperplane that separates the classes. These closest samples provide the *support vectors* of the method. The separation of the classes, with indication of the support samples, is illustrated in Figure 2.8b. A new, unlabeled sample is classified by whether it is above or below the hyperplane. If a linear classification boundary is insufficient for the data set, as in Figure 2.8a, we can perform a *kernel trick*. By elevating the samples in the input space to a higher dimensional space, we can search for a high-dimensional plane to achieve a linear separation. This illustrated in Figure 2.8.

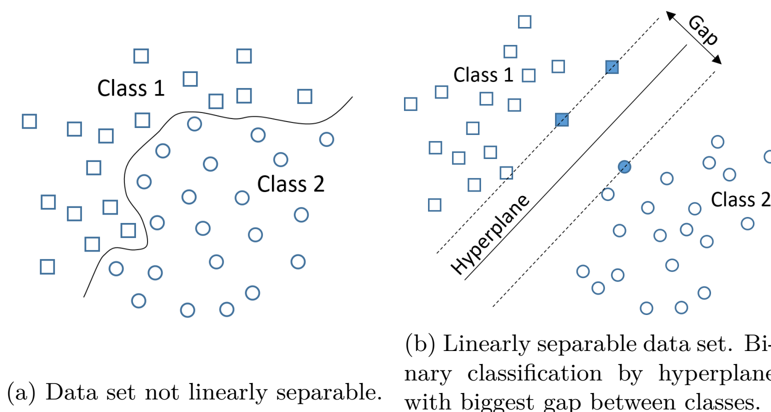
The SVM is an attractive classifier [16, p. 744], because:

1. It generalizes well, as it is based on a maximum separation (the hyperplane) between classes.

2. The kernel trick generates a linearly separable data set from a nonlinear.

3. The method is non-parametric

The use of SVM is not limited to only two classes. For multiple classes, it is possible to utilize a *cascade* of Support Vector Machines. This can be done for instance by representing each step as an "if-elseif-else" hierarchy. An object is either part of the first class, or it is not - and so the chain can go on. The size of this classifier will increase proportionally with the number of classes. Even though the method is not tuned with parameters, it will potentially store all the needed training examples (data samples) to generate the classifier. Nevertheless, as it obtains its classification ability from training examples only, it is like AdaBoost an off-the-shelf approach.



(a) Data set not linearly separable.

(b) Linearly separable data set. Binary classification by hyperplane with biggest gap between classes.

Figure 2.8: If a data set is not linearly separable, the multiplication with a kernel can transform it to a feature space where it is - e.g. from (a) to (b) in the figure.

Artificial Neural Network

An Artificial Neural Network (ANN) is a model created to mimic the structure of the human neural network. The nodes of a network is inspired by this biology, which is illustrated in Figure 2.9. A neuron, which is equivalent to *node*, has a number of inputs to represent the perception from a *dendrite*. The neuron processes the perception input, and determines if itself is *activated* to pass on a signal in the network. This neural process can be replicated with a simple mathematical model, and the concept of an artificial neuron is extended in Figure 2.10.

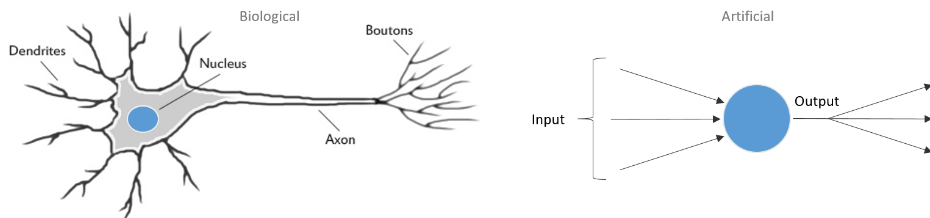


Figure 2.9: Illustration of a neuron. The nodes of an artificial neural network are inspired by biological neurons.

The mathematical model can be divided into three steps: (1) weighted input $w_{i,j}a_i$, (2) activation function g and (3) an output a_j. The activated output a_j from a node j, with a number of i inputs, can be described with Equation 2.7 [16, p. 728]:

$$a_j = g(\sum_{i=0}^{n} w_{i,j}a_i) \tag{2.7}$$

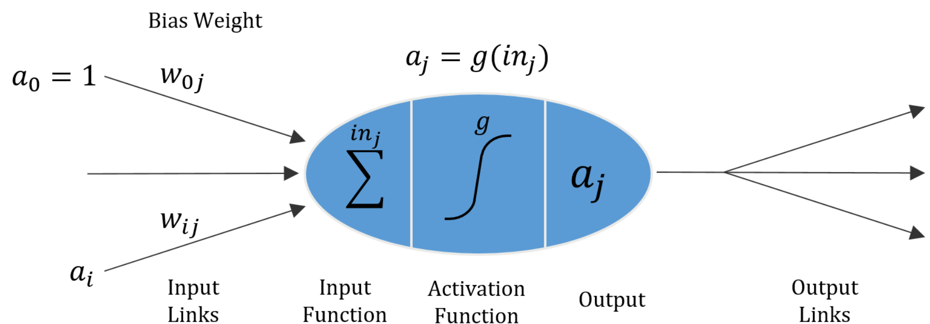
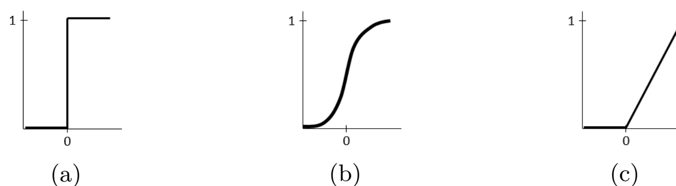


Figure 2.10: Simple mathematical model of a neuron. Illustration is based on figure in [16, p. 728].

For step (1), each input is the output a_i from another node. A node output is usually connected to multiple other nodes (as illustrated in Figure 2.12), which is why there is a unique weight $w_{i,j}$ from node i to node j. All of these activation inputs are accumulated in a sum. The activation output of the node j in focus is determined by an activation function g in step (2). Examples of typical activation functions are defined in Equation 2.8, and plotted in Figure 2.11.

$$\text{Threshold: } g_T(x) = \begin{cases} 0, & \text{for } x < 0 \\ 1, & \text{for } x \geq 0 \end{cases}, \quad \text{Sigmoid: } g_\sigma(x) = \frac{1}{1+e^{-x}}, \quad \text{ReLU: } g_{Re}(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases} \tag{2.8}$$



(a) (b) (c)

Figure 2.11: Example of activation functions: (a) Hard threshold, (b) sigmoid and (c) ReLU.

An activation function is used to determine if a node should contribute with a signal to the rest of the network, with an analogy to a neuron firing on perceptions. The hard threshold activation (Figure 2.11a), is a binary step (similar to a logical "OR" operator), the sigmoid activation is a soft step (Figure 2.11b), and the Rectified Linear Unit (ReLU, Figure 2.11c) rectifies the response by zeroing negative values. The choice of activation function for a ANN will depend on the purpose of the network, and the response that is required.

The final step, step (3), is to connect the output to other nodes in the network. A typical structure for both connecting and visualizing an ANN is illustrated in Figure 2.12. In a network like this, we group nodes into multiple *layers*. On a low level we have seen that a network can be customized with different activation functions (which is only one of many features to determine). From another point of view, an ANN model can be defined on a higher level by the number of nodes in each layer, the number of layers, and how the nodes are connected/linked together. As further illustrated in Figure 2.12, a network has an input layer, an output layer, and an optional number of *hidden* layers. All of these possible configurations, both on low and high levels, are what makes an ANN model attractive. Tuning of these parameters, and equally as important the *weights* of the node connections, provides great classification results without the designer

knowing how features are combined or weighted. The use of an artificial neural network is the "black-box" approach of the CV methods.

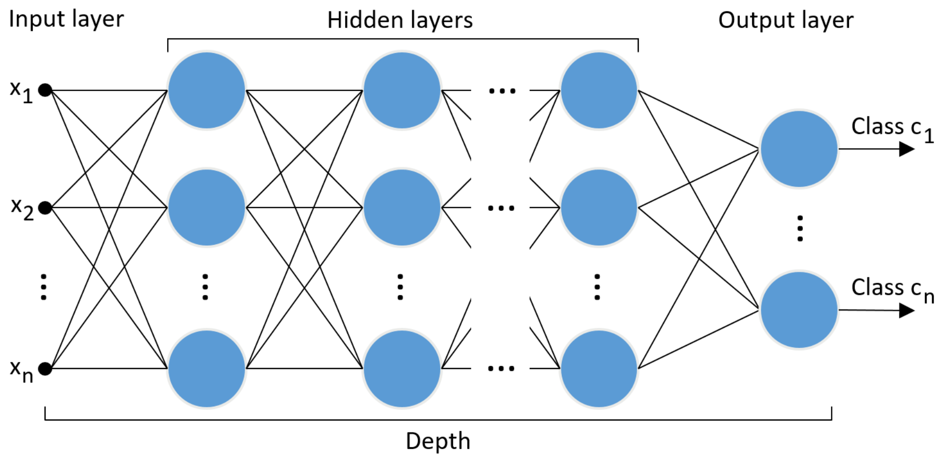


Figure 2.12: Typical structure of an Artificial Neural Network (ANN). The input layer is constructed from a pattern x, e.g. from an image patch. The output layer gives a confidence parameter c for each class. For this illustration, each hidden layer is fully connected, but this is optional.

Chapter 3

Computer Vision Application

Contents:
This chapter provides definitions on visual object detection and tracking, introduce the methods intended for the implemented application, and the most common performance metrics for detection and tracking. In addition to this, the final subsection defines a new hybrid tracking measure to evaluate tracker performance.

Literature review:
From the preliminary project [1] on CV, several surveys were used to present methods for object detection [6, 19–21] and tracking [7, 22–25]. The structure of CV as it is illustrated in Figure 1.2 is the result of these surveys. All of these papers are relevant for the discussion on the pipelines presented for both detection in subsection 3.1.3, and tracking in subsection 3.2.3. Some of the theory for these methods, as well as data set structuring, is also covered by literature used in the previous chapter [13, 14, 26].

For object detection, the survey "Object Detection: Current and Future Directions" [6] is central for object detection methods with assessment of their strengths and weaknesses. Verschae and Ruiz-del-Solar cites both classic studies for the relevant methods, as well as compare their work of this study with other object detection reviews. Out of these there is a review on detection of vehicles [19], a comprehensive review of the steps taken for object detection [20], and a survey on feature representation in statistical learning [21]. The most re-

cent paper used for the area of tracking is “A survey on moving object tracking using image processing” [22]. This paper introduces the basic steps taken in a tracking pipeline, and categorize tracking methods into sub-groups. The paper is complimented by older reviews [7, 23, 24], which all give similar presentation of the topic. The assessments presented in “Object Tracking: A Survey” [25], from 2006, also indicates that the sub-groups of tracking have remained the same for the past decade.

For tracking performance measures, the “Visual Object Tracking Performance Measures Revisited” [27] is a thorough elaboration on the most popular tracking measures in tracking literature. The critical focus on the intention and limitations of the measures is enlightening for further presentation of performance results.

3.1 Object Detection

This section is based on chapter 3 of the preliminary project [1].

3.1.1 Definition

Object detection is a sub-domain of the larger problem area *recognition* (as it was illustrated in Figure 1.2), alongside *instance recognition* and *class recognition* [14, ch. 14]. The simpler of these is *instance recognition*, which is to re-recognize a known rigid object (i.e. the same object) in a another environment. When we are looking to recognize *any* instance of a specific class, given the class is present, we are solving a problem of *class recognition*. Finally, the more intricate problem, *object detection*, is to determine the presence of a class and locate it:

> "Given a set of object classes, *object detection* consists in *determining the location and scale of all object instances, if any, that are present in an image*." [6]

In other words, object detection is the problem of finding and classifying all objects in an image.

3.1.2 Detection Method

To be suitable for the application area of the RWS, a detector must handle:

- Detection of multiple objects. This is because an operator must be aware of all possible threats.
- High confidence on its detections. Correct targets are crucial (and an obvious attribute) for the situations which require the use of a RWS.
- High accuracy of any detection. This is because the result from the detector will be used as the input for a tracker.

Methods for object detection can be compared qualitatively on accuracy, speed and generality. The motivation for a combined detector and tracker is that the detector should provide accuracy and generality to the system, which are strong attributes in deep learning based systems (as it was presented in subsection 1.1.3). The topic of deep learning is continued in subsection 3.1.4.

3.1.3 Pipeline

With the bottom-up description of image processing in section 2.1, there is an implicit algorithm in the process of object detection. These steps, from traversing an image to the presentation of a detection, can be visualized as a *pipeline*. By organizing the procedure in certain steps, it can be represented as in Figure 3.1.

Figure 3.1: Detection pipeline: Steps to take for object detection.

The first step, *candidate regions*, is what separates object detection from *class recognition*, as defined in subsection 3.1.1. Instead of an object covering the whole image (which makes the image the region), one or more objects are present in smaller areas of the original image. An intuitive approach would be to use a sliding window (explained in subsection 2.1.1), perhaps with different sized windows, to generate smaller image patches for the rest of the pipeline. This is not an effective approach, and not computationally sparse, but serves as an illustration of the concept. A more effective method for this step could be a Region Proposal Network (RPN), which is further explained in subsection 3.1.4.

The second step, *feature extraction*, is based on the concept explained in subsection 2.1.2 with the same name. This step is not only a step to extract features, but the features are also combined and stored to create some sort of representation or *descriptor*. The type of descriptor will depend on the application area, e.g. HOG for human detection [17]. The third step, *classification*, is what determines if the candidate region contains a given class or not. Depending on the desired response, this step (as with all methods) will depend on the application area. If the classification is binary, a SVM can be a good choice, even though it can be used in a cascade for multiple classes as well. Another option is to use an AdaBoost classifier, which together with the SVM is described in subsection 2.2.2.

The final step, *revise detection*, is more or less a key component of this detection pipeline. Together with the classification to determine if an object is present or not, the revision of this is to locate and output the position of the object. The output of the complete pipeline will depend on the methods used, and how the information is preserved along the line. If the input candidate regions are multiple neighbouring patches, the revision can consist of combining multiple similar detection results into a larger bounding box (which is a concept utilized by *single-shot detectors*, further explained in subsection 3.1.4). For a more precise

candidate region, the revision process can be simpler.

For the pipeline presented in Figure 3.1, it is important to notice that it is not a definite approach to a detection system. It is merely an illustration of the necessary steps for a detector to process, analyze and understand an image.

3.1.4 Deep Learning

With deep learning methods we have the advantage that representation can be transferred to other classes (transfer learning), and a common application area is search (as in e.g. retrieving information from pictures) [6]. The most notable downside for supervised learning, is the requirement of large data sets for training.

Deep learning is a domain within machine learning (supervised learning), entailing methods for learning data representations. Other than visual processing for CV applications, deep learning architectures can also be used for both speech and audio processing. This is because deep learning provides an Artificial Intelligence (AI) approach on such sensor problems, and mimics the human processing methods. The use of ANN models (see subsection 2.2.2) has accelerated the past years. Mainly, within CV, it is the variation Convolutional Neural Network (CNN) which has been most interesting - an architecture specialized for images. The use of CNN for face detection in 2004 [28] were one of the first successful studies for CV. Then, the interest for this method was brought back in 2012 with a classifier [29] showing great results in ImageNet Large Scale Visual Recognition Challenge (ILSVRC), compared to the other contestants.

Convolutional Neural Network (CNN)
A CNN is a stack of (i) *convolutional*, (ii) *pooling* and (iii) *fully connected* layers, and is a version of ANN customized for image inputs. The typical structure of a CNN is illustrated in Figure 3.2. A first obvious addition to the previously described ANN architecture, is the presence of convolutional layers. Several different filters are used to convolve a 3D input volume, a method described in subsection 2.1.1, to a 3D output volume of neuron activations. Each filter generates a feature map, and the convolution layer is the final stack of these. The reason behind the use of pooling layers (ii), is to down-sample the convolutional layer and reduce the number of model parameters (layer size). The idea is that a deeper network will create more advanced and aggregated features, that is, several separate features together creates a whole. It is assumed that this feature combination is more important than the resolution at each layer, which means that a layer can be down-sampled without much loss of information. Any suitable method for compressing the layer may be used, e.g. the average - *average pooling*,

or max values - *max pooling*.

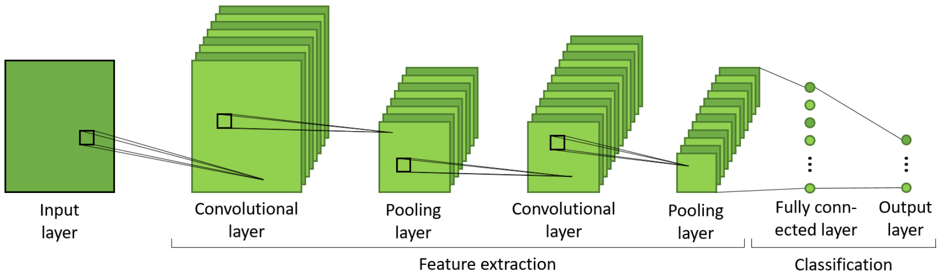


Figure 3.2: General structure of a Convolutional Neural Network.

The purpose of a final fully connected layer (iii), is to generate the output values. The fully connected layers will generate class predictions/ regression values from the features calculated by the convolution and pooling layers. As the name indicates, a fully connected layer is a layer with a connection between every node in the previous and the current layer. This means that a deep convolutional network will become very large if each layer is fully connected. With for instance an image, each layer will contribute with a number of $height \times width \times depth$ weights, which does not scale well for multiple layers. Only the final layers are usually fully connected in order to provide the predictions. For a number of c classes, an input image is transformed to a $[1 \times 1 \times c]$ vector of scores.

The depth of a convolutional layer is not to be confused with the depth of the complete network. In terms of this, the depth of the network is what provides the *deep* in deep learning. This depth, and the importance of it, is part of what made the classifier of 2012 [29] succeed. In this paper it is discussed that removing a hidden layer results in a notable loss of performance. Each layer in a CNN teaches the network different and more specified features to recognize objects.

Region based CNN

Girshick, now a part of Facebook AI Research (FAIR), has since 2013 been a continuous part of the teams to further develop the use of CNN. The basic concept of the method of Region based CNN (R-CNN) [31] is to provide region proposals to a CNN, to limit the search area of the image. With this paper it was also presented that *supervised* pre-training of the network provided better performance, and that CNN outperformed systems with simple HOG-like features (method presented in subsection 2.1.3). The application of CNN for object detection is obvious with the use of R-CNN, as it outputs *bounding boxes* around objects in addition to provide class labels. The region proposals are generated by the use of

selective search [32]. For a high level understanding, the image is searched with different sized windows, and each window try to group adjacent pixels based on texture, color, or intensity to identify objects. The final step of the R-CNN, is to use a SVM to determine if the window patch contains an object or not - and then classify the object.

A couple of years later, Girshick published a paper on *fast* R-CNN [30]. This paper proposed a simplification of the original R-CNN, which resulted in both faster processing and easier training. The architecture of the first R-CNN is designed to send image features through the network, also called *forward pass*, for each window proposal in every image. Additionally, all the models (CNN, the SVM, and a regression model for the bounding boxes) need separate training. For faster R-CNN, the forward pass is reduced to only *once* for each image. This is done by selecting the Region of Interest (RoI) from the corresponding CNN feature map, and not from the input image. Then the features from each region is pooled, which is called *RoIPool*. The training issue is solved by combining the previous three models into one. This is done by replacing the SVM classifier with a *softmax*, or sigmoid-normalization (see the sigmoid function in Equation 2.8), on top of the CNN. Parallel to this, a linear regression layer outputs bounding boxes.

The final iteration of the R-CNN improvement, called *faster* R-CNN [33], identifies the region proposal model as the bottleneck of the *fast* method. As the region proposals from the selective search are dependant on the features from the CNN [30], the region proposals are instead embedded into and proposed from the CNN feature map. Another CNN uses the feature map to propose regions, creating a Region Proposal Network (RPN) within the system - a process illustrated in Figure 3.3. With this, the only input to the faster R-CNN is an image, and the output is class predictions and bounding boxes for object detection. As concluded in the paper: “By sharing convolutional features with the down-stream detection network, the region proposal step is nearly cost-free” [33].

Single-shot detectors

The name Girshick can also be found in the publication of the “You Only Look Once” (YOLO) architecture [34]. This model can be defined as a *single-shot detector*. Like the faster R-CNN model, the input is only an image which is “forward passed” once through a CNN, but the RPN is omitted. Instead, the input image is pre-divided into an $S \times S$-grid (RoI), and bounding boxes with associated classes are predicted within each grid. With class predictions for each grid element, a class probability map can be generated. Adjacent grid elements with the same highest class probability is combined to generate the final bounding box for an object.

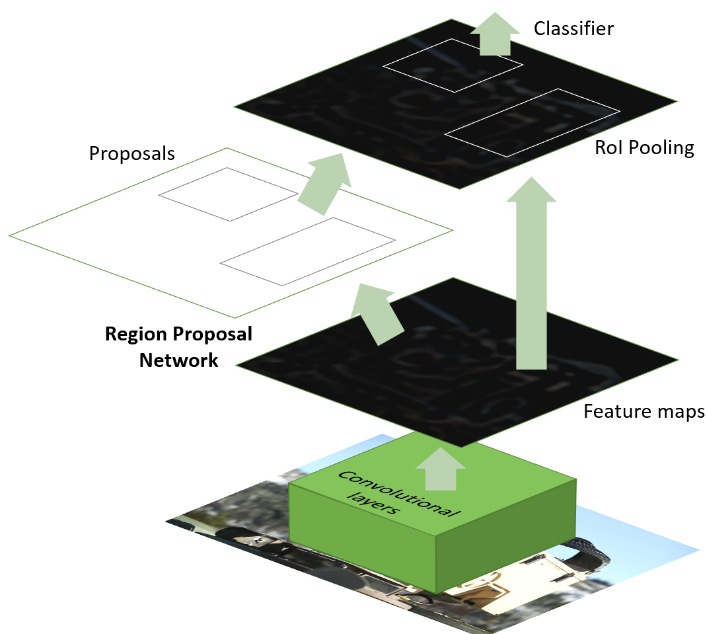


Figure 3.3: Faster R-CNN. A unified network for object detection, with a CNN and a RPN. The illustration is based on a similar figure in [33].

A competing architecture to YOLO, is the "Single Shot MultiBox Detector" (SSD) [35]. The first notable difference in the pipeline, is that a pre-defined grid is used to divide the output space - not the input image. The SSD architecture uses a set of boxes, with different sizes, determined by training. At run-time, the class probabilities are calculated for each of these boxes in output space, which are adjusted to create the final bounding box to fit an object. Secondly, SSD uses multiple feature maps for detection, compared to the use of only one map in the YOLO model. The SSD feature maps also decrease in size to adjust to the different sized output boxes. The paper points out that "[SSD] is faster than the current real time YOLO alternative" [35]. However, this was before the publication of "YOLOv2" [36] with improved performance.

3.2 Object Tracking

This section is based on chapter 4 of the preliminary project [1].

3.2.1 Definition

Computer Vision (CV) can be separated into areas of research based on problem types, as it was illustrated in Figure 1.2. However, this does not mean that the research areas *detection* and *tracking* are completely unrelated. Tracking can be defined with the following formulation:

> "Tracking is the problem of generating an inference about the motion of an object given a sequence of images." [13, p.326]

That is, when the position of an object is given in the first image, a tracker is desired to find and follow the same object through the rest of the consecutive images. As with object detection, to be able to track an object, we need features to describe the image and methods for segmenting and classifying interesting regions. The difference between these two lies in the preliminary information. A detector is supposed to locate *any instance* of a specified object when it appears in an image. A tracker, on the other hand, is supposed to find the *same instance* of any object specified from the initial image. In other words: a detector uses information gathered from other sources, and a tracker uses the information given in the image sequence (e.g. video) at hand.

The information inferred from an object at different frames can be stored as an object *state*. This state variable can include one or more of the relevant attributes (in different combinations): position, velocity, acceleration and appearance [13]. What information to extract, and keep track of, will depend on application area and available resources.

3.2.2 Tracking Method

To be suitable for the application area of the RWS, a tracker must handle:

- Video with *dynamic background*. This is because of either movement of the vehicle the RWS is mounted on, or movement of the RWS itself.
- Objects of *different sizes*. The possible objects of interest, drone, person and vehicle (as presented in subsection 1.1.3), represent all object sizes from small to large.
- Object representation for a variety of shapes.
- Object occlusion
- Real-time processing of the input video.
- Tracking of a single target to assist the operator.

Methods for object tracking can be compared qualitatively based on how occlusion, entry/exit of objects and multiple objects are handled. For a tracking system to be able to handle objects with different physical attributes, it is assumed that a point based method will be suitable (as stated in subsection 1.1.3). This is especially because of the ability to track small objects. A fast tracker may be subject to *drift* (gradually shifting away from the target), which motivates the desire to periodically update the track with a more accurate detection. The topic of point based tracking is continued in subsection 3.2.4.

3.2.3 Pipeline

As for detection in subsection 3.1.3, tracking can also be presented as a pipeline. With background in the representation presented in the surveys [7, 22, 23, 25], the steps for a tracking method can be illustrated as in Figure 3.4

Figure 3.4: Tracking pipeline: Steps to take for object tracking.

However, this representation is not unanimous. Another view on simple tracking strategies is to divide them into *tracking-by-detection* and *tracking-by-matching* [13, ch. 11]. For the first case, as with the approach in the surveys [22, 23, 25], the strategy is based on a strong model of the *object*. This model is strong enough to identify the object in each frame. The second case, tracking-by-matching,

is used when there is a model of how the object *moves*. When we know the region of which an object is located in one image frame, we use the motion model to find a region in the next frame. This region is then subject to a matching model, a way of comparing the previous domain in frame n with the domain in frame $(n + 1)$. The matching model is used to determine that the object is the same. This alternative division is to somewhat supported by what is presented in [24], with the difference of dividing the strategies into tracking-by-detection and *statistical tracking*. With this second opinion on the tracking pipeline, the pipeline in Figure 3.4 can be extended and illustrated as in Figure 3.5.

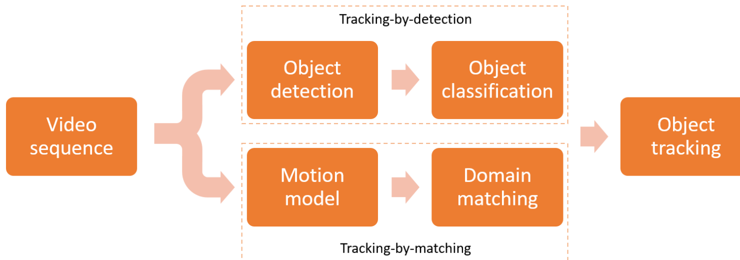


Figure 3.5: Extended pipeline, split in steps for either tracking-by-detection or tracking-by-matching.

The tracker is intended to follow objects in consecutive images, which is why the first step is to input a video source. For *object detection* in this pipeline, there are several possible methods. It is stressed that there is a difference to how types of detection methods for tracking are classified [24], based on other surveys. From this, we can see that the oldest survey [25] group methods into (i) background subtraction, (ii) point detector, (iii) segmentation and (iv) supervised learning. However, newer surveys [22, 23] rather use the grouping (in addition to (i) background subtraction) (v) frame difference and (vi) optical flow.

Both background subtraction (i) and frame difference (v) are closely related, as well as intuitive. With frame difference, the difference between two frames are calculated to indicate any movement. This is easy to implement, but it is difficult to extract an accurate location for a moving object. The background subtraction is a bit more complex to implement. With (most conventionally) a stationary camera, the background can be subtracted, and the remaining information is thresholded and segmented to obtain the moving object. Because of the benefit of a stationary camera, this is a popular method for security applications [24]. Segmentation (iii) can also be used on its own. This is a basic image operation which labels regions in an image, based on e.g. color, shape or pattern. It can

for instance be easy to segment out the squares on a chess board based on color. The motivation for a good segmentation operator lies in both efficiency, and how to chose a segment.

The method of optical flow (vi), is more related to a tracking-by-matching approach. The *optical flow* is the resulting apparent motion in an image, caused by scene or camera motion relative to an object [16, p. 939]. A measure of similarity is calculated, to match a new domain in frame $n+1$, by e.g. the sum of squared differences as in Equation 3.1:

$$\text{SSD}(D_x, D_y) = \sum_{(x,y)} (I(x,y,t) - I(x+D_x, y+D_y, t+D_t))^2 \tag{3.1}$$

The optical flow at (x_0, y_0) is $(v_x, v_y) = (\frac{D_x}{D_t}, \frac{D_y}{D_t})$, which is measured in [pixels/hour]. The last two methods for this, (ii) point detection and (iv) supervised learning, are not completely independent. Point detection is still only extraction of feature points (see subsection 2.1.2) either from distinct positions or intensity values. But to find a matching point in a sequential frame, a calculation of for instance optical flow is needed. Supervised learning is more of a technique for data training, and detectors based on this requires large sets of data (as previously mentioned).

In the final step of the pipeline in Figure 3.5, *object tracking*, a model of the object's motion is calculated. The more frames that pass, the stronger the object model should get. This can be stored in an object state variable, and the increasing information gathered on the object can be used to improve the target tracking. The tracking-by-matching part of the pipeline, which can be seen as statistical tracking, have the advantage of being faster then the other strategy. This is because tracking-by-matching is mostly based on motion prediction, and does not require the system to execute a detection process on each frame [24].

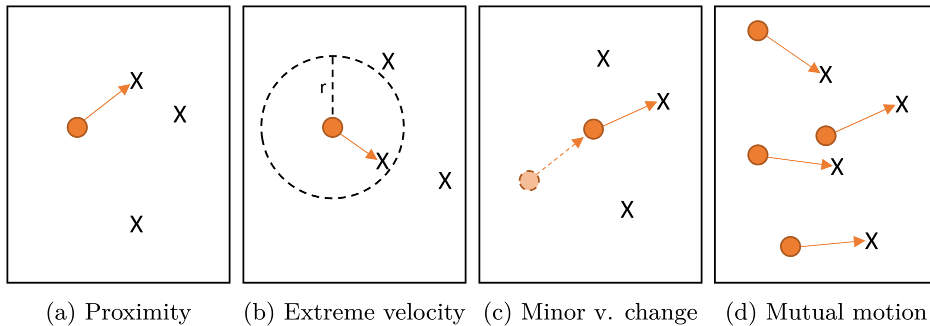
3.2.4 Point Tracking

Point tracking is a method where an object is represented by its feature points (see subsection 2.1.2), and corresponding points are tracked between frames. The points are associated by the previous object state with position and motion. Because of the point representation, this category is appropriate for tracking small objects, but it is required to use an external mechanism to detect the object in each frame.

Finding corresponding features from one frame to another comes with a cost [7, 25], and can be determined by a combination of the following assumptions:

- **Proximity**: The location will not change much in the next frame.
- **Extreme velocity**: There is a boundary on max velocity of an image point.
- **Minor velocity change**: The velocity of a point does not change much.
- **Mutual motion**: The velocity of points in a neighbourhood tend to be similar, as well as the distance between points in a group (when the object is rigid).

These assumptions are illustrated in Figure 3.6.



(a) Proximity (b) Extreme velocity (c) Minor v. change (d) Mutual motion

Figure 3.6: Illustration of the kinematic limitations of point tracking. The circle indicates a point in frame $n-1$ and X a point in frame n. These illustrations are based on figures in [7, 25].

The assumptions can be used to calculate an optimal path both for deterministic and statistical tracking methods. The deterministic approach would be to use a greedy selection of points, e.g. a shortest path algorithm, to chose corresponding points based on the assumptions. The method of using *Kalman filter* for tracking

is a statistical approach. It is widely used in CV for noisy environments, all the way back to 1986 [37]. As it assumes the state variables to be Gaussian distributed, it will result in poor performance if it is not [25]. The track generation is based on prior and posterior knowledge of the state variables. This is generated in a two-phase manner: prediction and correction, illustrated in Figure 3.7, with the predicted state being corrected from measurements in the next frame. This limitation of the Kalman filter is overcome by using a *particle filter*, which also use a two-phase procedure for the state, but weigh the importance of a sample by its appearance frequency.

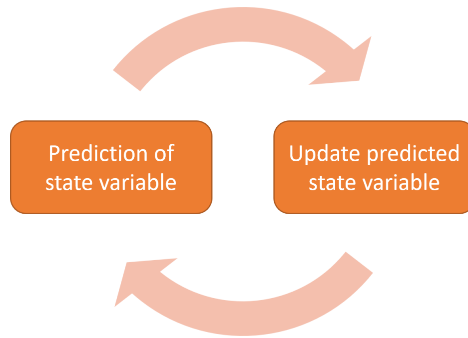


Figure 3.7: The two-phase process of updating the state variable, used by Kalman filtering.

A last example on a point based tracking method is Multiple Hypothesis Tracking (MHT). As indicated by the name, it creates multiple hypothesis for the position of each feature point. On a frame to frame basis, the model of the motion will become increasingly better, and the most probable trajectory is the one to track [22]. This method can maintain multiple suggestions at a time, and thus create new and remove old ones in time with objects entering and exiting the Field of View (FOV). A prediction for the position of a point will be calculated for each frame, and compared to a distance measure for detection. With multiple objects, and then several hypothesis, this method can be computationally heavy for a large set of sample points. However, the scalability of such a method will not be an issue as long as the application is single target tracking.

3.3 Training and Evaluation

3.3.1 Data Structuring

This subsection is based on parts from subsection 2.4.2 of the preliminary project [1].

With a deep learning approach there is a continuous notion on the importance of large data sets. Deep learning methods, compared to methods with predefined and manually defined features of interest (e.g. the methods in subsection 2.1.2), extract the knowledge of relevant features from the input data it is provided. For a human, it is possible to determine that an object is a car - even if both brand and model is not seen before. This is mostly because the daily interaction with cars of different types and models. A similar (and maybe more intense) approach is used for training of a neural network: the network is repeatedly exposed to a number of images, and learn object characteristics from examples.

The size of the data set, and how it should be utilized, will depend on the system model and the intended application area. The data set should be divided into two or three parts: mainly two separate *training* and *testing* sets, and optionally a *validation* set from a portion of the training set. To be perfectly clear, these three sets can be defined as [26, p. 354]:

- **Training set:** A set of examples used for learning, that is to fit the parameters of the classifier.
- **Validation set:** A set of examples used to tune the parameters of a classifier, for example to choose the number of hidden units in a neural network.
- **Test set:** A set of examples used only to assess the performance of a fully-specified classifier.

The separation of the training and test set is important for the integrity of the performance results from model evaluation, and to prevent any bias from peeking at some of the test samples during training: "Peeking is the consequence of using test-set performance to both *choose* a hypothesis and *evaluate* it. The way to avoid this is to *really* hold the test set out - lock it away until you are completely done with learning and simply wish to obtain an independent evaluation of the final hypothesis" [16, p. 709]. A validation set can be used to measure performance on unseen data, and with a test set "locked away" the remaining data is divided into training and validation sets. An illustration of this can be seen in Figure 3.9a.

In addition to tuning of model parameters, the validation set is an important

measure to oversee the training process. After a certain number of training cycles, the model will start to specialize on features and characteristics only present in the training set, and the performance on the validation will therefore become weaker. This phenomenon is known as *overfitting* or over-training, and can be illustrated as in Figure 3.8 in terms of the training error on the output predictions of the model. There are a few regularization methods for dealing with the problem of overfitting:

(i) Early stopping

(ii) Dropout

(iii) Cross-validation

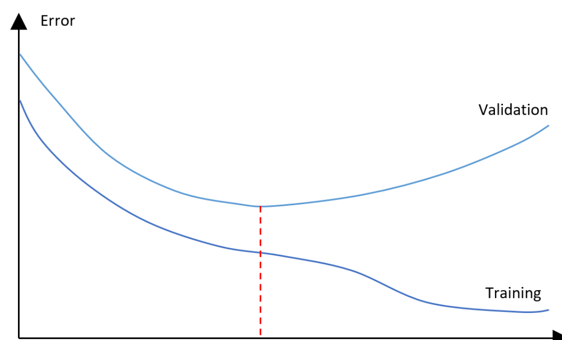
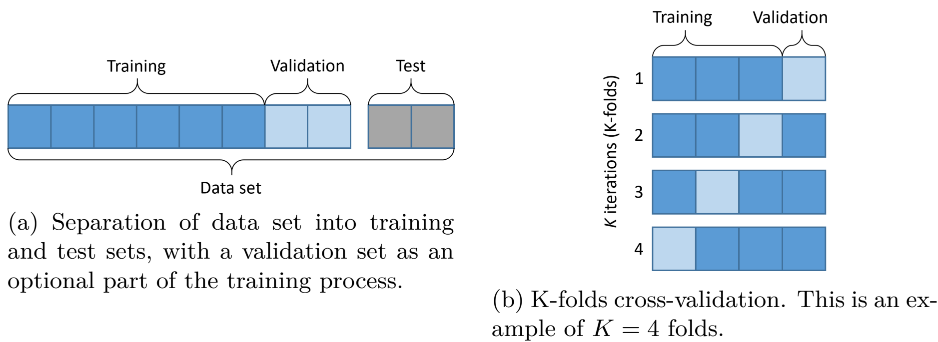


Figure 3.8: Overfitting: After a certain number of training cycles, the model will start to specialize on the training set. This will decrease the general performance, as illustrated in an increasingly worse training error for the validation set.

With *early stopping* (i), the idea is to stop the training as soon the error on the validation set is higher than when previously checked, and use the previous weights of the model [38, p. 56]. This method can prove to be unfortunate if the stopping point is not the global minimum. Checkpoints during training can also be stored, for a final evaluation of the validation error after a fixed number of training cycles. The use of *dropout* (ii) can significantly reduce overfitting and give major improvements over other regularization methods [39]. The idea is to randomly drop nodes and their connections throughout the network, which prevents nodes from co-adapting too much and maintain node activity. Finally, the effect of a validation set can be increased with *cross-validation* (iii) [13, p. 464]. This is illustrated in Figure 3.9b, which includes splitting the data set into separate training and validation sets differently, a number of K times. The resulting validation error is the average error of all the K sets.



(a) Separation of data set into training and test sets, with a validation set as an optional part of the training process.

(b) K-folds cross-validation. This is an example of $K = 4$ folds.

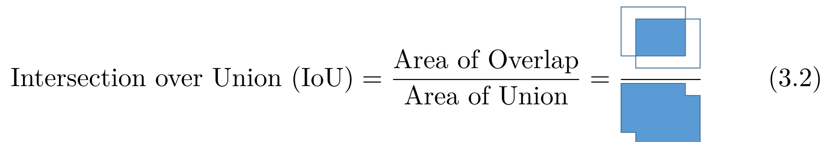
Figure 3.9: Data set separation.

3.3.2 Performance Measures

This subsection is based on parts from subsection 2.4.2 of the preliminary project [1].

Detection

When evaluating a method or technique for detection, there are a few terms used to describe the result. A detector should locate and determine the size of an object, and this is usually done with a *bounding box*. A bounding box is a rectangle which creates a boundary around the object, and is usually desired to have a close fit to the outer edges of any object. An intuitive measurement to determine accuracy during training, when evaluating performance, is the Intersection over Union (IoU) for the prediction. The predicted bounding box is compared to a ground truth bounding box for the object in the image, resulting in a value between 1 and 0. An IoU is illustrated in Equation 3.2, which yields that a perfect fit will result in a value of 1. It is then possible to determine the average IoU of multiple detections, for a total evaluation.



$$\text{Intersection over Union (IoU)} = \frac{\text{Area of Overlap}}{\text{Area of Union}} = \tag{3.2}$$

A good detector does not only need to output accurate locations, but also have to be confident on its classifications. When using a data set with both positive and negative cases (e.g. images with and without a given class), we can define the *accuracy*, *recall* and *precision* of a detector. To determine this, the prediction is seen as a binary choice; either the class is present in the image or it is not.

This results in four outcomes for each detection (prediction of an object), shown in Table 3.1:

Table 3.1: Prediction outcomes for object detection.

	Positive case	Negative case
Positive prediction	True positive (TP)	False positive (FP)
Negative prediction	False negative (FN)	True negative (TN)

The terms are then defined (with all positive and negative cases denoted PC and NC) as:

$$accuracy = \frac{(\mathrm{TP}+\mathrm{TN})}{(\mathrm{PC}+\mathrm{NC})}, \quad recall = \frac{\mathrm{TP}}{\mathrm{PC}}, \quad \text{and} \quad precision = \frac{\mathrm{TP}}{(\mathrm{TP}+\mathrm{FP})} \tag{3.3}$$

Out off all the ratios defined in Equation 3.3, accuracy is one of the more intuitive measures. To put it in words, accuracy is a ratio on how many predictions were correct for all cases. The recall is a ratio on how many of the positive cases were predicted, and the precision is the number on how many positive predictions were actually correct.

A standard metric for search and prediction algorithms is either the average precision (AP) or **mean average precision (mAP)**[40], which utilize both *precision* and *recall* as they are defined in Equation 3.3. As the name states, it is a double average of precision values, but it might be a bit misleading for how it is actually calculated. A more suitable name may perhaps be “order-matters recall” (quote by Zygmunt Zając, `fastml.com`), because correct predictions early in a test sequence gives a higher score. This metric is best explained with a custom example, as provided in Table 3.2.

First of all, the average precision (AP) is calculated from a fixed sequence of predictions, but the precision sum is only taken of the true positive (TP) predictions. The way the recall value is used, is by *change* in recall (here the notation $\Delta recall$ is used) from one sample k to the next. The AP is calculated as the precision multiplied with the change in recall. With no change in recall for a FP *only the TP predictions contribute to the value.* The two prediction sequences shown in Table 3.2 illustrate how the average precision is better if the true positives occur earlier in the sequence, which means that the order matters. The final mean value is calculated as an average of the total test sequences - and is usually noted as a mean value “at” k ($mAP@k$).

Table 3.2: An example on calculation of mean average precision at different k-values, to illustrate the $mAP@k$ value. In this batch there are 10 test images, with a total of 5 positive cases (PC). With a positive prediction in every image, there will be equally many true positive (TP) and false positive (FP) predictions distributed in the test batch. Two different prediction sequences are provided to illustrate how the order of the TP predictions matters for the AP and the final mAP value. The average precision is calculated as the sum of ΔRecall $\cdot$ Precision.

k	1	2	3	4	5	6	7	8	9	10
Prediction:	TP	FP	TP	FP	TP	FP	FP	FP	TP	TP
Recall:	0.2	0.2	0.4	0.4	0.6	0.6	0.6	0.6	0.8	1.0
Δ Recall	0.2	0	0.2	0	0.2	0	0	0	0.2	0.2
Precision:	1.0	0.5	0.67	0.5	0.6	0.5	0.43	0.38	0.4	0.5
$AP@k$	0.2	0.2	0.33	0.33	0.45	0.45	0.45	0.45	0.54	0.64
Prediction:	FP	FP	TP	TP	TP	FP	TP	FP	TP	FP
Recall:	0	0	0.2	0.4	0.6	0.6	0.8	0.8	1.0	1.0
Δ Recall	0	0	0.2	0.2	0.2	0	0.2	0	0.2	0
Precision:	0	0	0.33	0.5	0.6	0.5	0.57	0.5	0.56	0.5
$AP@k$	0	0	0.07	0.17	0.29	0.29	0.4	0.4	0.51	0.51
$mAP@k$	0.1	0.1	0.2	0.25	0.37	0.37	0.43	0.43	0.53	0.58

Tracking

The paper "Visual Object Tracking Performance Measures Revisited" [27] is a survey on visual object tracking performance measures. The survey presents a thorough collection of the most popular measures for single-target visual tracking, and states that "none of them is a *de-facto* standard". This serves as a good starting point for considering relevant performance measures, and the different approaches for performance assessment are listed as: (i) center error, (ii) region overlap, (iii) tracking length, (iv) failure rate, (v) hybrid measures and (vi) performance plots. The paper [27] also establishes a general definition of an object state description Λ in a sequence with length N frames:

$$\Lambda = \{(R_t, \mathbf{x}_t)\}_{t=1}^{N} \tag{3.4}$$

In Equation 3.4, $\mathbf{x}_t \in \mathcal{R}^2$ denotes the center of the object and R_t the region/area of the object at time t. For this thesis the region R_t is described by an axis aligned bounding box, and the state for the ground truth box and the predicted box is denoted Λ^G and Λ^T.

The center error (i) is a measure of distance between the center of the predicted target and the the center of the ground truth. This measure requires minimal annotation preparation for a data set, with just a single point in each frame. The center can also be derived from a ground truth bounding box. The Euclidean distance, the shortest straight line between two points, is used to calculate the distance error between the prediction $\mathbf{x}_t^T$ and the ground truth $\mathbf{x}_t^G$. This measure does not take the size of the bounding box into account, but the drawback can be improved by normalizing the distance with rights to the box size. A discussion of the center distance error is continued in subsection 3.3.3, with a proposition of a new distance score.

The region overlap (ii) is the same measure as the IoU as defined in Equation 3.2. With the the same notation as in Equation 3.4, a region overlap ϕ_t at time t, and the *average overlap* $\bar{\phi}$, can be written as:

$$\phi_t = \frac{R_t^G \cap R_t^T}{R_t^G \cup R_t^T} \quad \textit{and} \quad \bar{\phi} = \sum_t \frac{\phi_t}{N} \tag{3.5}$$

The region overlap is also used in combination with a threshold, say $\tau = 0.5$, to calculate a success rate - the number of successful predictions out of all the processed frames N. Both tracking length (iii) and failure rate (iv) are also measures usually dependent on thresholds. The length of tracking can be determined by the number of continuous successful predictions, and the failure rate is the number of *not* successful predictions, which are rates filtered by thresholds. The

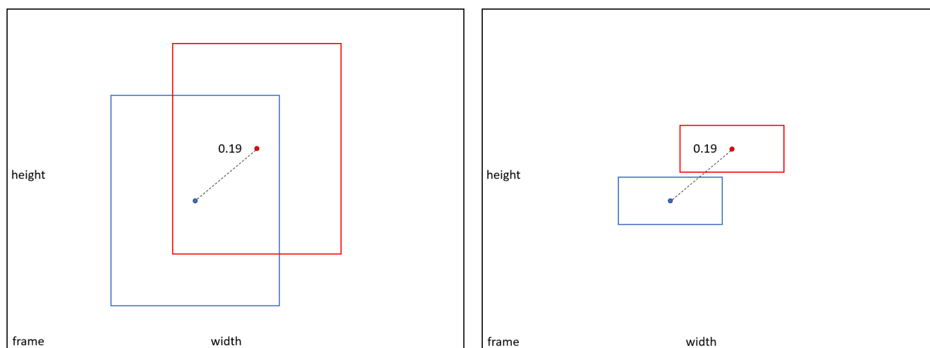
papers [41–43] are examples of experimental comparison of several algorithms, which use the success rate from the overlap and the average center location error to evaluate performance on several well-known videos. This combination of overlap and center error is widely used [27]. A perspective on the use of threshold for success rate is presented in [42], which states that a static threshold may not be fair or representative for the tracker. A score derived from the area under chart (AUC) is proposed as an alternative, but this is proven to be the same as the average overlap value $\bar{\phi}$ in Appendix B of [27].

The final two approaches, hybrid measures (v) and performance plots (vi), are usually just utilization of the basic measures. This is to present more robust evaluations, and simpler visualizations of the performance results. In terms of visualization, the most common approaches are to plot the center error or the region overlap, with respect to the frame number.

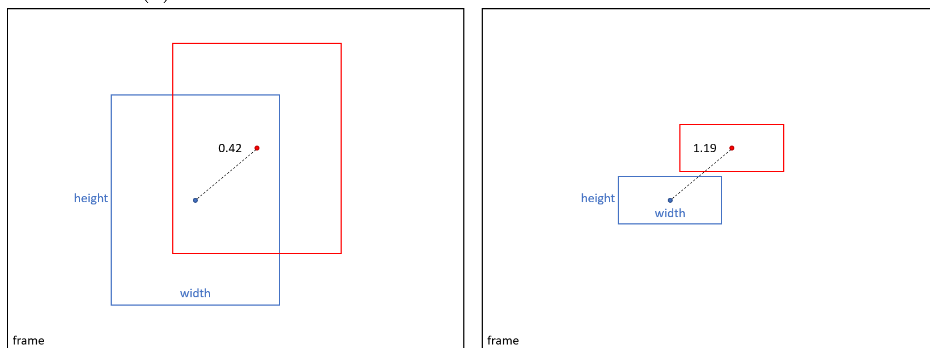
3.3.3 Proposition of a Hybrid Tracking Measure

Center Distance Error Issue
The calculation of the center distance error, as it is described for tracking in subsection 3.3.2, will have two different approaches (as stated in [27]). When calculating this distance relative to the frame size, the measure completely ignores the target's size (size of the bounding box) and does not reflect the apparent tracking failure. An illustration of this measure in situations with boxes of different sizes can be seen in Figure 3.10a. With the same resulting Euclidean distance for both situations, there is no intuitive indication that one of these predictions completely miss the target (the situation illustrated to the right).

The second approach, intended to remedy this, is to normalize the distance error with rights to the area of the ground truth bounding box. This approach is illustrated in Figure 3.10b. With this measure, it is easier to notice that one situation depicts a better prediction. Still, this approach is dependent on some kind of threshold to determine the meaning of the error. A further introduction to a new *center distance score* is provided, for use in this thesis, as an attempt to improve the center distance measure and its intuitive meaning.



(a) Euclidean distance with reference to the size of the frame.



(b) Euclidean distance with reference to the size of the ground truth bounding box.

Figure 3.10: Illustration of two approaches on the calculation of the center distance error. The frames depict two situations with the same physical distance between a blue ground truth box and a red prediction box, but with different amount of box overlap. The center distance error is calculated as the Euclidean distance in normalized coordinates. In (a) it is calculated with reference to the image frame, and in (b) with reference to the ground truth box.

Proposed Center Distance Score

Continuing the state notation from Equation 3.4, the components can be defined as:

$$R_t = w_t \cdot h_t \quad and \quad \mathbf{x}_t = (x_t, y_t) \tag{3.6}$$

With Equation 3.6, the center error in both x and y direction can be defined as in Equation 3.7:

$$\delta_{t,x} =\mid x_t^G - x_t^T \mid \quad and \quad \delta_{t,y} =\mid y_t^G - y_t^T \mid \tag{3.7}$$

With Equation 3.6 and Equation 3.7, a score $S_\delta \in [0, 1]$ (relative to the ground truth region R_t^G) can be defined to calculate the accuracy of the predicted center position:

$$S_{\delta,t} = 1 - min(max\left(\frac{2\delta_{t,x}}{w_t^G}, \frac{2\delta_{t,y}}{h_t^G}\right), 1) \tag{3.8}$$

An illustration of the center distance score S_d for a prediction, as it is defined in Equation 3.8, can be seen in Figure 3.11. The ground truth bounding can be considered a shooting-target, and the distance score is determined from the maximum distance error in either x or y direction. A predicted center $\mathbf{x}_t^T$ right on top of the ground truth center $\mathbf{x}_t^T$ will result in the max score $S_\delta = 1$. This score will decrease with an increasing error, until the boundary with $S_\delta = 0$.

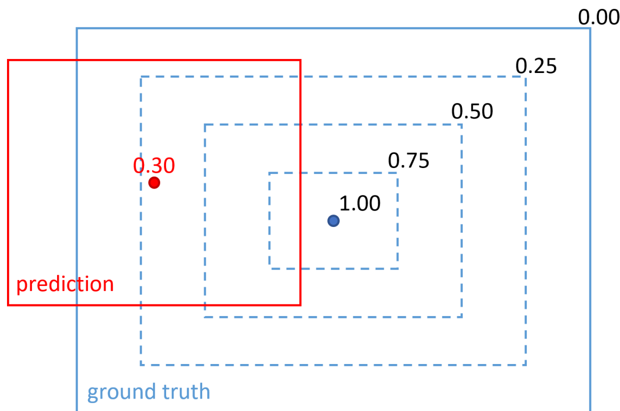


Figure 3.11: The distance score $S_\delta = 0.30$ for a predicted bounding box. The score S_δ is determined by the greatest distance deviation in either x or y direction between the box centers, and calculated relative to the center of the ground truth box. $S_d \in [0, 1]$ with score 0 for any center outside the bounds of the ground truth box, and score 1 for a complete overlap.

Hybrid Measure for Tracker Performance
The Combined Tracking Performance Score (CoTPS) is a combination of tracking accuracy and tracking failure in a single score. The definition of this in Equation 3.9 [27] is only provided for reference purposes, with $\bar{\phi}$ for the average overlap and λ_0 percentage of failure frames (when $\phi = 0$):

$$CoTPS = 1 - \bar{\phi} - (1 - \lambda_0)\lambda_0 \tag{3.9}$$

This approach is criticized by the paper on tracking measures [27], and it is stated that "[...] two very different basic measures are being combined in a rather complicated manner, prohibiting a straightforward interpretation". Taking this into account, an attempt on using a more intuitive combination of measures will be further presented.

The position of the bounding box as a *center error* and the size of the box in terms of *region overlap* will be interesting to evaluate for the purpose of the RWS. Both the center of the target (which for most cases is the same as the center of the bounding box around a rigid object), and the object size are highly relevant for a RWS. With the center distance score proposed in Equation 3.8, the center error is defined in terms of the same range $[0, 1]$ as the IoU. These two (now similar) measures can be combined in a straightforward manner, with an average:

$$S_{T,t} = \frac{1}{2}\left(S_{\delta,t} + \frac{R_t^G \cap R_t^T}{R_t^G \cup R_t^T}\right) = \frac{1}{2}(S_{\delta,t} + \phi_t) \tag{3.10}$$

The hybrid tracker performance score S_T, as it is presented in Equation 3.10, is ment to balance two popular basic measures into a single intuitive result. In the tracking measure survey [27] it is pointed out that a center error normalization alone may give misleading results, because the center error is reduced proportionally to the object size. A large bounding box may result in a smaller error than a bounding box with the correct size. This may be prevented when considering both the region overlap and position in a single hybrid score S_T. The intention of introducing a new hybrid measure for this thesis, is to quantify both the size and position in a single bounding box metric, and treat these measures as equally important when evaluating an object tracker for a RWS.

Part II

Implementation

**"Ideas are easy,
implementation is hard."**

Guy Kawasaki

Chapter 4

Software

Contents:
This chapter is a discussion of available software frameworks and data relevant to the detection and tracking application. It also highlights different aspects for consideration when relying on open source software.

Resources:
To limit the scope of possible resources for a CV implementation, decisions should be made with focus on possibilities for the Jetson TX2 module and the use case of the RWS. With this, both the Jetson TX2 hardware and software capabilities will play an important role in how an autonomous tracking application is developed. A lot of the references provided in this chapter will be to online resources, because of the dynamic nature of software and how it is distributed. Note that *inference*, i.e. reaching a conclusion on basis of reasoning, is a term used for the object detection process.

The Jetson TX2 has the following hardware specifications:

- NVIDIA Pascal™ Architecture GPU
- 2 Denver 64-bit CPUs + Quad-Core A57 Complex
- 8 GB L128 bit DDR4 Memory
- 32 GB eMMC 5.1 Flash Storage
- Connectivity to 802.11ac Wi-Fi and Bluetooth-Enabled Devices
- 10/100/1000BASE-T Ethernet

All the Jetson platform specific software is bundled together in *JetPack 3.2* [44] with L4T (Nvidia Linux for Tegra driver package), which includes:

- **TensorRT**: Speeds up deep learning inference as well as reducing the runtime memory footprint for convolutional and deconv neural networks.
- **cuDNN**: CUDA Deep Neural Network library provides high-performance primitives for all deep learning frameworks. It includes support for convolutions, activation functions and tensor transformations.
- **CUDA Toolkit**: CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications. The toolkit includes a compiler for NVIDIA GPUs, math libraries, and tools for debugging and optimizing the performance of applications.
- **GStreamer**: GStreamer is a library for constructing graphs of media-handling components. The applications it supports range from simple Ogg/Vorbis playback, audio/video streaming to complex audio (mixing) and video (non-linear editing) processing.
- **OpenCV**: OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision.

4.1 Frameworks and Supportive Software

4.1.1 Deep Learning Framework

First of all, the term framework is here understood as an abstraction for a reusable software environment, which provides standard functionality for a specific task or application purpose. This means that a *deep learning framework* is software that provides support for deep learning methods, such as construction of a neural network for visual object detection. As it is explained for both artificial neural networks in subsection 2.2.2, and the different modifications of it for object detection in subsection 3.1.4, there are a lot of mathematical operations and algorithms in play. Instead of having to implement all of these basic operations from scratch, a framework provides - just like a toolkit - assisting functionality and building blocks for design of applications.

Relevant Attributes
Because of all the possible options and combinations of available software to use, it can be useful to consider what attributes or functionality that are necessary for the application. For the development of a novel deep learning based system, the focus should be on criteria that can help current and future implementation. The following attributes could be considered, with focus on the Jetson TX2:

- Support for GPU acceleration
- Support for TensorRT
- Versatile and easy to use interface
- Framework with optimized core implementation
- Support community
- Cloud computing
- Licensing

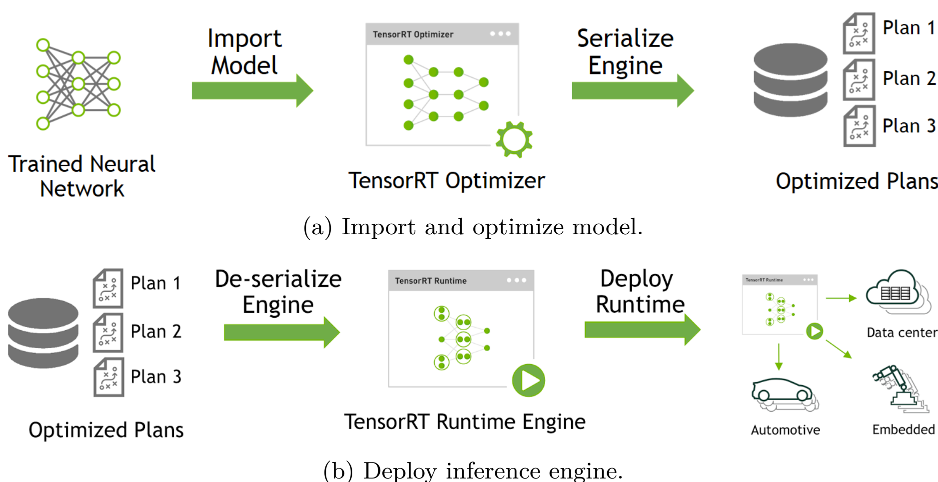
In the recent years, multiple deep learning frameworks have been developed for deep learning purposes, and a few of these have already managed to manifest themselves as popular. A list of popular deep learning frameworks with GPU support is presented in Table 4.1, which only show a few of the possible options. Nvidia also provides a list [45] of frameworks that are compatible with their own GPU acceleration, which include: *Caffe2, Cognitive Toolkit, MATLAB, MXNet, NVIDIA Caffe, PyTorch, TensorFlow, Chainer* and *PaddlePaddle*. This is important as both CUDA and cuDNN [46] are included in the Jetson TX2 software bundle, as mentioned in the beginning of this chapter.

Table 4.1: The most popular, open source deep learning frameworks with support of GPU acceleration. The frameworks are listed alphabetically.

Framework	Developer	Interface
Caffe	Berkeley Artificial Intelligence Research (BAIR)	C, C++, Python, MATLAB and command line
Caffe2	Facebook	C++ and Python
Keras[1]	Francois Chollet	Python
Microsoft Cognitive Toolkit (prev. CNTK)	Microsoft	Python, C++, C# and command line
MXNet	Apache Software Foundation	Python, R, C++ and Julia
TensorFlow	Google	C++ and Python[2]
Theano	Université de Montréal	Python[2]
Torch	Ronan Collobert, Koray Kavukcuoglu and Clement Farabet	C, C++ and Lua

[1] Python API to other back-end deep learning frameworks.
[2] Support Keras front-end.

TensorRT
Cross-referencing the Nvidia list with the one in Table 4.1, a lot of the frameworks are still standing. For the software interface, Python can be considered as an easy to use and flexible scripting language, but it is important that the core implementation is effective. This is usually the case with hardware near implementations like C/C++, and therefore a combination of these languages would be preferable. All of the frameworks in the Nvidia list are also compatible with TensorRT (listed in the beginning of this chapter). With TensorRT, the inference process for the Jetson TX2 can be optimized. This work-flow is illustrated in Figure 4.1, with (a) presenting the steps intended on a host computer, and (b) the steps on the target/embedded platform.



(a) Import and optimize model.

(b) Deploy inference engine.

Figure 4.1: The TensorRT work-flow. From training and optimization on a host computer (a), to generated inference engine on a target platform (b). The illustration is taken from Nvidia's description of TensorRT [47].

The intention of developing an application aimed at TensorRT is to fulfill the assumption in section 1.3 which state that *embedded deployment will benefit from a [...] "best practice" use of the third-party tools utilized.* When considering the use of Python, it is important to notice that at the time of writing the Python Application Programming Interface (API) for TensorRT is not supported on the Jetson TX2. For now, this can make it difficult to create a streamlined application in Python for the embedded platform.

Because almost all of the mentioned frameworks can seem to be suitable for

the use with the Jetson, and the work-flow presented for TensorRT, future use cases should be considered as well. This will include consideration of a possible support community, how versatile it is, and the possibility for cloud computing. Cloud computing could be a contributing factor, because of e.g. an economic perspective. This is because dedicated hardware for supervised learning is rented instead of owned. With support in Google Cloud, Google's TensorFlow framework is guaranteed cloud support. For a future notice on technology, TensorFlow will be interesting because of Google's development of Tensor Processing Units (TPUs) [48]. With the GPU introducing parallel processing, the TPU introduces custom tensor processing. The TPU is an integrated circuit for AI applications specifically designed for neural network machine learning. This does not mean that any of the other frameworks are not supported by such a service, or any other alternative like Amazon Cloud or Microsoft Azure. However, it will ultimately depend on on the most relevant approach for the future.

TensorFlow

The abstraction TensorFlow uses for neural networks is in the form of a *data flow graph*. Each node in the graph represent mathematical operations, like the illustration of a neuron in Figure 2.10. The edges (all the pathways between the neurons) represent multidimensional data arrays called *tensors*. TensorFlow also comes with a suite of visualization tools called *TensorBoard* [49], which can be used to visualize models and data. This can definitely be an important tool for training, for example in the case of preventing overfitting as described in subsection 3.3.1 with the visualization in Figure 3.8.

For the implementation of the autonomous tracking system, Google's *TensorFlow* deep learning framework is further used as a starting point. There is no definite answer for why none of the other frameworks are chosen, as many of the frameworks seem to be compatible with development for the Jetson TX2. Still, TensorFlow fulfills all the relevant attributes presented in the beginning of this section. And as a starting point for object detection, Google's object detection API [50] in Python seems like a good candidate. This is also because of the available state-of-the-art neural network models provided for object detection, further discussed in subsection 4.2.3.

License

The deep learning frameworks listed in Table 4.1 are all open source, but their software rights are all specified by one of three licenses: Apache 2.0, MIT or BSD. A summary of the relevant permissions, conditions and limitations for these licenses are listed in Table 4.2, Table 4.3 and Table 4.4. Considering future production of a RWS vision system built with TensorFlow, the corresponding Apache

2.0 includes permissions for commercial use, distribution and patent use - which are all relevant use cases from a business perspective.

Table 4.2: License permissions

License	Commercial use	Distribution	Modification	Patent use	Private use
Apache 2.0	√	√	√	√	√
BSD 3	√	√	√	-	√
MIT	√	√	√	-	√

Legend: √ Yes, X No, - Not stated

Table 4.3: License conditions

License	License and copyright notice	State changes
Apache 2.0	√	√
BSD 3	√	-
MIT	√	-

Legend: √ Yes, X No, - Not stated

Table 4.4: License limitations

License	Liability	Trademark use	Warranty
Apache 2.0	X	X	X
BSD 3	X	X	X
MIT	X	-	X

Legend: √ Yes, X No, - Not stated

4.1.2 Tracker Implementations

For object tracking, when focusing on a method based on more traditional CV, there are no common frameworks developed. Usually, tracking algorithms are implemented for the purpose of a specific task or application area. To not be occupied with implementation of the steps of a tracking algorithm from scratch, it will be easier to find a relevant tracker implementation. Such an implementation should be a type of *point based* tracker, as suggested by the preliminary project [1] presented in subsection 1.1.3.

Referencing the software list included in the beginning of this chapter, OpenCV (Open Source Computer Vision Library) is part of the Jetpack 3.2 package. This is a commonly known CV library which is supported in Python, and there is also an existing object tracking API under development. The object tracking API [51] is currently not part of any official release/version of OpenCV, but it can be accessed through a *contribution* package (opencv-python-contrib). The API provides use of five implementations of different tracking algorithms, with the one called *MedianFlow* showing the highest frame rate count. The implementation of this algorithm is based on the paper that first presented the Median Flow tracker [52], which is a tracking algorithm that tracks feature points from an initial region of interest. The MedianFlow-algorithm is the most promising implementation, with respect to the *point based* approach desired for the tracking module.

The use of OpenCV, as it is a library specifically made for computer vision, will make it possible to implement custom tracking algorithms for future use. This should serve as a good staring point when designing a modular application, with the possibility of changing the tracking method without changing the code which utilize it.

4.1.3 Security

In subsection 2.2.2, specifically in the description of ANN, it is stated that the use of an artificial neural network is the “black-box” approach of the CV methods. This is because it is difficult to pin-point exactly what layer combinations, connections and features that work best for a use case - and it is also difficult to evaluate risks if not all factors are known.

A relatively recent paper [53] uncovered intriguing properties of neural networks. It is possible to manipulate an image in a way imperceptible to humans, and make a deep neural network label an object as something completely different to what it actually is. The paper describes how “adversarial examples” are made by optimizing the input to maximize the prediction error. This means that, in con-

trast to optimizing the weights of the network to minimize the prediction error (for an accurate prediction), the input image itself is changed to cause a wrong prediction. By using limits on how much the intensity values in the image can change, the manipulation can be made imperceptible to the human eye.

Another paper [54], researching a similar issue, shows how easy it is to construct images completely unrecognizable to humans that are positively labeled. Images looking like white noise or meaningless patterns can be classified by state-of-the-art deep neural networks as specific objects - with a *99.9% certainty*. Both of these papers show how this is mostly accomplished with access to the model of the neural network, to tailor the images for the system it is intended to fool. This can become an even greater issue if a system is based on a publicly available detection model, such as the ones provided by TensorFlow. In theory, distinct printed patterns can then be used to camouflage objects from being detected by a future autonomous RWS vision system. Future risk assessments of vision based systems will possibly be subject to new neural network properties.

4.2 Available Computer Vision Data

4.2.1 Benchmark Data Sets

Deep learning methods are dependent on data to train models, and for the detector it will be important to train with relevant images. With the popularity and increasing research on CV methods, attempts are continuously made to create standard data sets to evaluate performance. Some of the most recognized *benchmarks* for deep learning in CV literature are:

- *ImageNet* [55] for image classification
- Microsoft *COCO* [56] for object detection
- Visual Object Classification *VOC* [57] for image classification

These benchmarks include large amounts of data, in form of images and annotations (ground truths), for training of CV systems. There are also more specific data sets in other benchmarks, depending on the CV application, but the three listed are known for their large number of different classes. Models trained on either of these data sets will be relevant for the RWS detection module, because classes such as *airplane*, *person* and *vehicle* are all included. This will enable direct use of a trained model, or it can serve as a base for transfer learning of more specific custom classes like drones or military vehicles.

The same goes for object tracking. Even though the tracking method chosen does not require information about data prior to deployment, compared to the training process for deep learning, test videos are still important. Videos, or sequences of consecutive image frames, with annotations are necessary for evaluation. As with detection, there are also some interesting benchmarks for object tracking:

- Computer Vision Lab *CVLab* [43] with publicly available data sets [58]
- Visual Object Tracking Challenge *VOT* [59] with publicly available data sets [60]

The CVLab data set is a collection of videos used in multiple other tracking papers, with both short term and long term tracking videos, and both single and multiple objects. This is also the case for the VOT data set, but it is also used for an annual challenge. Both of these data sets include different scenarios relevant for multiple application areas. An important disadvantage when using these sets, is that none of them contain any videos of aerial objects. For testing purposes this will require custom data construction.

4.2.2 Custom Test Videos

It is difficult to find relevant annotated drone data. The data available for a more general aerial object/class, such as *airplane*, is also scarce. A solution to this is to create new test videos from sections of YouTube videos, and make relevant video clips with representative object behaviour. This is still with focus on airplanes, because most available drone videos are from the drone's point of view. Table 4.5 lists the four videos created, with URL to the original videos and which time slots of the originals that were used. A thumbnail for each one of these videos can also be seen in Figure 4.2.

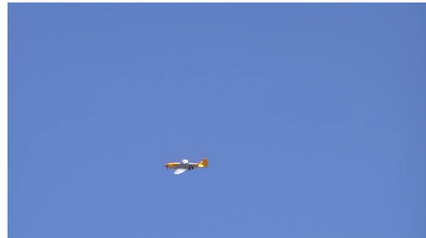
The custom videos in Table 4.5 all include different scenarios that can prove important during the development process of an autonomous tracking application. The first video, *BlueAngels*, is a video of four fighter jets flying in formation. With multiple objects in the video, it is possible to decide how the system (initially intended for single target tracking) will deal with this issue. The video section is also subject to video clipping (in the original video), and thus objects will fade, reappear, and can suddenly change both position and angle. This is also the case for the last one in the list, *ToyPlane*. Only a single target is visible, but there are a lot of scene changes and overlaps. In this video the airplane is also rather small.

The video *RCBoeing747* is the most stable and continuous of the custom videos. This is a video of a passenger airplane replica, flying in a big circle parallel to the ground. With this yaw rotation, both the object silhouette and size will change. The final video, *HobbyKing*, is perhaps the most interesting. This is because the original video is of a model plane, and in this section of the original video the camera is zoomed in on the airplane in flight, with very unstable camera handling. Because of this, the video section includes irregular motion, object out of focus, change in size, and rotations in multiple directions - with turning, loops and rolls. This is also the only video with manually created annotations. Ground truth data is important for the evaluation of the final tracking application, but the manual process is time-consuming. A 20 seconds short video, with a capture rate of 30 fps, will consist of 600 individual images.

Table 4.5: Custom test videos

Video	Source	Time [min:sec]
BlueAngels.mp4	`https://youtu.be/LkrnpO5v0z8`	05:27 - 05:54
HobbyKing.mp4	`https://youtu.be/XhUzoIm6OQo`	00:54 - 01:14
RCBoeing747.mp4	`https://youtu.be/akoJ2zBwX1o`	01:15 - 02:45
ToyPlane.mp4	`https://youtu.be/y8G2ezOmz1w`	06:06 - 07:47

(a) BlueAngels



(b) HobbyKing

(c) RCBoeing747

(d) ToyPlane

Figure 4.2: Images of the four custom test videos listed in Table 4.5.

4.2.3 Deep Learning Models

With reference to subsection 3.1.4, the most recent and best performing models for object detection are considered to be *faster R-CNN* and *single-shot detection* (SSD). For the object detection module, a model based on either of these methods will be interesting to deploy.

In the *TensorFlow detection model zoo* available on GitHub [61], a collection of pre-trained models is provided for "out-of-the-box" inference. The relevant models for the detection module are listed in Table 4.6. Because of the continuous update of this model zoo, the models are most relevant for the system at the time of writing; the list is likely to be updated with new and improved models. The listed models are pre-trained on the COCO (Microsoft Common Objects in Context) data set [56], containing images with a total of 91 different object types. As addressed in subsection 4.2.1, a model pre-trained on this data set will be interesting with the relevant classes included. For custom data or classes, it can be used as a base for transfer learning.

Table 4.6: Object detection models, relevant for the detection module, from the TensorFlow *detection model zoo* [61], and pre-trained on the COCO data set. The reported running time is in *milliseconds* (ms) on a 600 × 600 image, with a *Nvidia GeForce GTX TITAN X* GPU.

Model name	Speed (ms)	Speed (fps)	COCO mAP
faster_rcnn_inception_v2_coco	58	17.2	28
ssd_inception_v2_coco	42	23.8	24
ssd_mobilenet_v2_coco	31	32.3	22
ssd_mobilenet_v1_coco	30	33.3	21

In Table 4.6, both of the names *faster_rcnn* and *ssd* are familiar from subsection 3.1.4. The name *mobilenet*, on the other hand, refers to a different architecture. The *MobileNetV2* is a new mobile architecture [62], which show improvements in mobile networks. With the paper presenting conclusive notes on increased performance and reduction of model parameters, it can be a good choice for an embedded platform with limited resources. The table is first of all sorted by the highest *mean average precision* mAP (see subsection 3.3.2), but the trade-off in speed is also apparent with a decreasing processing time down the list. For the detection module of the tracking application, a higher mAP is desired, but there will still be limitations on the processing speed to consider when choosing a model. This is because of the system design when combining a detector and a tracker, and will be further discussed in subsection 5.1.2.

Chapter 5

System Implementation

Contents:
This chapter describes the system design and architecture for a command line application implemented in Python. This includes presentation of the intended work-flow, limitations and the final application interface.

Resources:
The system implementation is made publicly available on GitHub [9], and the structure for the repository *detection-and-tracking* can be seen in Appendix A.

The following frameworks are used:

- TensorFlow: Google's *object detection* API [50]
- OpenCV: Tracking API [51], which is a part of the *OpenCV-contrib* package

Custom implementations in Python for this thesis:

- Detection module/class which utilize the TensorFlow object detection API
- Tracking module/class which utilize an OpenCV tracker
- Video capture module/class for asynchronous video capture
- Tracking application to combine the above modules
- Scripts to calculate performance results

Additional scripts implemented in Python/Bash to automate and streamline processing of data:

- Download test images, and create playable videos
- Normalize ground truth annotations
- Create annotations for new videos (manual process)
- Calculate relevant measures (overlap and error distance) from the tracking results
- Installation and environment setup for the tracking application

System structure:
The implemented system is structured in a hierarchy of folders as displayed in Listing 5.1. The system is partitioned into top levels *host* and *target* for software with intended use on respectively a desktop computer and the Jetson TX2, and the custom videos listed in Table 4.5 are stored in *videos*.

```
detection-and-tracking/
|--- host/
     |--- scripts/
     |--- src/
          |--- detector/
          |--- tracker/
          |--- utils/
     |--- test/
          |--- cvlab/
          |--- vot2017/
|--- target/
     |--- scripts/
|--- videos/
     |--- data/
```

Listing 5.1: System folder hierarchy.

5.1 Design

5.1.1 Approach

The Real-time Constraint

To understand the issue and why a combination of a detector and a tracker is interesting, an example of a detector alone can first be considered. If a hypothetical detector (with an arbitrary hardware configuration) use a time $t = 0.1s$ to process a single image, this translates to processing $f = \frac{1}{t} = 10$ frames per second (FPS). If there are no restrictions on either the detector or the input video, each frame from the video can be captured at a pace suitable for the detector. The problem, however, may present itself when real-time restrictions are introduced. The definition of live camera video in section 1.3, with a capture rate of 30 FPS, will create a conflict with the hypothetical detector. With this input rate, the detector will only be capable of processing 10 out of 30 frames each second, which means that a number of frames will be skipped periodically; in this case two.

The introduction of a tracker is intended to cope with this real-time constraint of live video. A "light weight" tracker, which only considers features provided online (i.e. at system runtime with no pre-processing of data), have the ability to process frames at a *much* higher rate. But with high speed, there is usually not much consideration of features, which results in a not so robust tracker. This can for instance mean that the tracker will *drift* away from the intended target, and rather track something else in the background. The *combination* of these two modules is the main idea for complying with the real-time constraints: a fast tracker updated periodically by accurate detections. When the detection module is supposed to find any instance of an object for a general approach, the tracker is only intended to find the specific object provided by the detector. This idea is illustrated in Figure 5.1. The blue sequence indicates the rate of which new frames are captured, and the numbers are used to illustrate which frame the detector and tracker is processing at each new frame capture.

Frame Buffer

When a detector skips frames, this does not only mean that there are frames not being processed, but the results from the processed frames will be old. In other words, the detector will bring latency to the system. The effect of a slow detector will depend on the application area, but with a moving object the object might not be anywhere near the detection result after a few frames. This is a key issue with such a combination: the detection provided for the tracker must be accurate and relevant for the current frame.

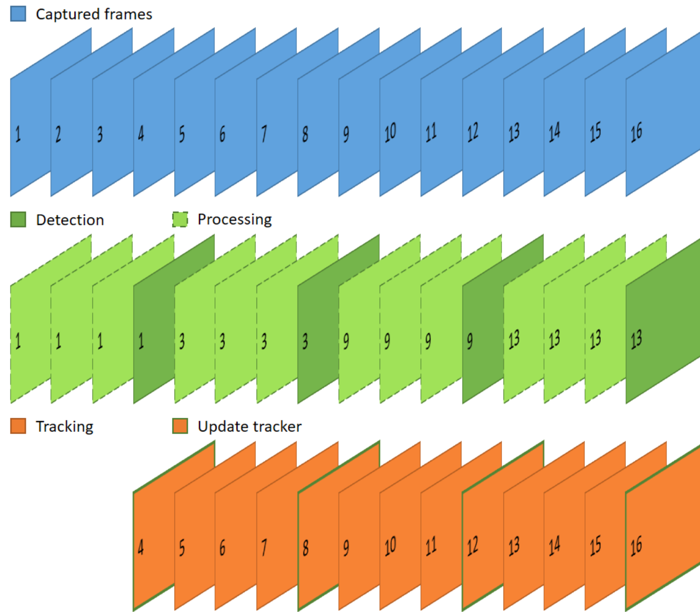


Figure 5.1: Illustration of a frame processing timeline. The top sequence is the consecutive frames captured from camera video, with the detector and tracker below. The numbers indicate which frame is being processed at each step.

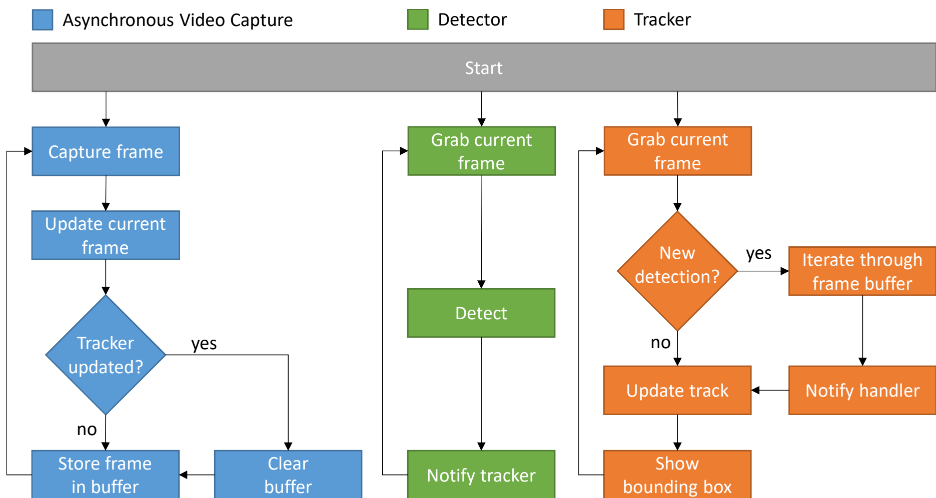


Figure 5.2: System decision tree and module interaction.

To fulfill this requirement, a *frame buffer* is introduced. By storing the skipped frames between each detection, the oldest frame in the buffer will correspond to the detection result. This means that the tracker can be initialized with this frame, track through the frames in the buffer, and predict an up-to-date result for the current frame. An illustration of this algorithm is presented in Figure 5.2, with frames stored in a buffer, and the tracker being corrected periodically with notice of a new detection. This approach means that the tracker does not only have to process the captured frames, but also an additional number of frames for each detection. This is a system limitation further discussed in subsection 5.1.2.

5.1.2 Limitations

An important assumption, which is the foundation for this design, is that the tracker has to process frames fast enough to iterate through the frame buffer before a new frame is due. That is, the tracker must be able to process a number n of frames in the time period t_v between frames in the video. This requirement on the processing period of the tracker t_t can be defined as in Equation 5.1:

$$t_t \geq \frac{t_v}{n} \tag{5.1}$$

These variables can be further expressed in terms of the frequency measurement FPS:

$$t_t = \frac{1}{FPS_t}, \quad t_v = \frac{1}{FPS_v} \quad \text{and} \quad n = \frac{FPS_v}{FPS_d} \tag{5.2}$$

Equation 5.2 also shows that the buffer size n is a number on how many frames that go by in between detections, which is dependent on the video rate. The real time requirement on the system is defined by the capture rate of the camera video, and thus it is more natural to denote the speed dependency in terms of FPS. With the notation in Equation 5.2, Equation 5.1 can be formulated as:

$$FPS_t \geq \frac{FPS_v^2}{FPS_d} \tag{5.3}$$

Equation 5.3 indicates that this way of combining a detector and a tracker creates a notable dependency on the processing speed of both modules. With a trade-off in accuracy and speed on the detector part, an accurate detection will require more processing time. However, with the dependency illustrated in Equation 5.3, there is a finite limit on how accurate the detector can afford to be. And the other way around, as they are inversely proportional, the tracking method has to be fast enough to process the frame buffer before a new frame is captured.

5.2 Implementation

5.2.1 Modules

The implementation of the tracking application in Python is object oriented, and for the time of writing intended on the host platform. As it is illustrated in Figure 5.2, there are three modules implemented: *VideoCaptureAsync*, *Detector* and *Tracker*. The detector and the tracker modules are the main components discussed throughout this thesis, and the video capture module is developed to provide more control over both this and future video source scenarios. The modules are described with input arguments and available class methods.

VideoCaptureAsync (Python Class):

- **File**: src/videocapture.py
- **Dependency**: opencv-python
- **Arguments**: source, width, height and fps.
- **Methods**: start, wait, read, read and clear frame buffer and stop.

This class requires OpenCV for the internal read functionality of the input *source*, and the frame *width* and *height* can be specified. The additional *fps* argument is to provide control for testing. The module is started with the *start* method, which creates an independent thread for video capturing. This thread is dependent on an internal time thread, which signals a periodic "tick"-event according to the *fps* argument, and realizes the specific video capture rate. This is the asynchronous part of the video capturing module, with the current frame stored periodically. From the application program it is also possible to read the current frame, *wait* for a new frame (triggered by an event) and both read and clear the *frame buffer* (as described in subsection 5.1.1).

Detector (Python Class):

- **File**: src/detector/detector.py
- **Dependency**: object_detection (TensorFlow API)
- **Arguments**: capture source, model, labels, number of classes.
- **Methods**: start, wait and stop, as well as *get* methods for detections, fps, class ID and class name.

This class requires the TensorFlow Object Detection API to run inference. The *capture source* is the VideoCaptureAsync module, for the ability to read the current frame. The neural network *model* used for object detection (e.g. a model listed in Table 4.6) must be specified, with additional model specifications such as *labels* for object classes and the *number of classes* to detect. As with the VideoCaptureAsync module, the detector is also created in an independent thread with *start*. That is why additional class methods are necessary to read the *detections* and other relevant status information.

Tracker (Python Class):

- **File**: src/tracker/tracker.py
- **Dependency**: opencv-python-contrib
- **Arguments**: tracker type.
- **Methods**: init, update, get bounding box and get fps.

This class requires the OpenCV Tracking API (which is under development). The tracker module can be seen as a wrapper, as it basically extends the tracking API with the *init* and *update* methods. The tracker module uses the *tracker type* MedianFlow as default (as mentioned in subsection 4.1.2). One interesting drawback of the OpenCV implementation of the tracking algorithm, is that it is only possible to initialize a tracker *once*, which means that it is required to create a new Tracker class instance to re-initialize the object tracker. This is necessary for the ability to evaluate new detections, and utilize the frame buffer.

5.2.2 Application Work-flow

The application is intended to mimic the original module design as it is presented in Figure 5.2. With this said, there are some modifications to how the modules signal each other in a more implicit way. Both the video capture module and the detector module run in separate threads. This leaves the tracker module to be used explicitly in the tracking application **tracker-app.py**.

The capture module only updates the current frame and store the previous one in a buffer. Any resetting of the frame buffer is done by an external method call. The tracker does in a way notify the video capture module that it has used the frame buffer, but this is only by calling the *clear buffer* method. For the detector, it does not notify the tracker explicitly, but a status variable can be checked when fetching the recent detections - to determine whether the detection is old or new.

A section of the source code from **tracker-app.py** in Listing 5.2 illustrates the key algorithm for the combination design. For each new (and positive) detection, the video capture buffer will be read (and cleared), and a new tracker will iterate through the frame buffer to transform the detection to a prediction in the current frame. If not, the tracker is just updated normally with the current frame.

```
79 # Tracker update
80 if new_detection:
81     if bbox_d:
82         time_d = time.time()
83         buffer = cap.read_frame_buffer()
84         if buffer:
85             tracker = Tracker()
86             tracker.init(buffer.pop(0), bbox_d)
87             for f in buffer:
88                 tracker.update(f)
89     else:
90         cap.clear_frame_buffer()
91 tracker.update(frame)
```

Listing 5.2: Key section for tracker algorithm.

This “tracker update algorithm” in Listing 5.2 is constrained to run only once for each new captured frame, to avoid unnecessary polling and use of resources. This is realized with the *wait* method for the capture module, which triggers an event for each new frame. In order to synchronize the use of the frame buffer, and prevent that the detector read and process a frame which later will be deleted from the buffer before a detection is ready, the wait method is used in the detection module as well. After the detector is finished with a detection, it will wait for the next frame.

For the design limitation as expressed in Equation 5.3, no explicit system action is performed if the constraint is not complied with. Nevertheless, the processing rates are monitored, and information is printed for the user in the terminal:

$$\text{[!] Warning: FPS_t} = FPS_t \text{ is too slow (limit} = \frac{FPS_v^2}{FPS_d}\text{).}$$

Here with placeholders FPS_t, FPS_v and FPS_d for the real values calculated at runtime. This is provided to indicate that the detection model is too slow for the current hardware configuration (assuming the tracker “model” *MedianFlow* is default).

5.2.3 Post Processing of Raw Predictions

The modules described in subsection 5.2.1 utilize third-party interfaces (detection and tracking API), but the raw prediction data is further processed in the **tracking-app.py** implementation to serve the purpose of the system. These operations can be listed as the following *post processing* steps:

1. Filter out single best detection based on object class and confidence threshold
2. Throw away prediction if time-out on detection occurs
3. Stabilize the tracking prediction by considering the size of previous bounding boxes

These steps are somewhat products of continuous testing of the implementation during development, by running object tracking on the custom videos described in subsection 4.2.2. These videos, as previously stated, all include different scenarios that may be important for the use case of a RWS. The handling of the scenarios by the raw tracker will therefore be relevant pointers to what attributes the final tracker should possess.

Filter object detection
To be clear, step 1 is only one way - a more or less greedy way - of handling the reduction in number of objects. As it is specified in section 1.3, the implementation is intended to explore a *single target* tracker. The CNN models used with the TensorFlow object detection API are specified to output multiple detections to be able to detect a larger number of object in each frame, even multiple instances of the same object type. The issue of data association in object tracking is a field of its own, and thus filtering out the best detection from the relevant target class is probably the easiest way out. This method, however, creates complications when introducing the tracker to a video with multiple same-class objects - such as in the *BlueAngels* video. The result of this is that the tracker will jump between the different objects, as soon as the confidence score is higher for another object than the previous detection. With this greedy filtering step, the tracker does not only work for single object tracking, but it will not produce reasonable predictions when multiple objects are present in the same frame.

Consider tracker integrity
For step 2 it is all about connecting the confidence and accuracy of the detector to the prediction of the tracker. In addition to keep track of an object in between detections, and iterate through missed frames presented in the frame buffer, the tracker can still maintain track of an object even if the detector for some reason

does not recognize an instance. There are both positive and negative sides to this ability, and the video *ToyPlane* indicates how it can be a problem. This video has several overlapping clips with the airplane fading in and out of view, change in illumination, and a lot of background objects with distinct features (e.g. buildings). When the camera is following the airplane, the background objects will be moving relative to this in the opposite direction. This will in some cases make the tracker drift away from the target, and continue to track background objects instead. With step 2, a timer is used to put a constraint on such undesired behaviour. By resetting a timer for each new positive detection, a time-out threshold can be evaluated for each tracking prediction: if it has been too long since the last detection, the tracking prediction looses its integrity, and it should no longer be presented. The intention of the tracking application is to extend accurate detections, and thus the prediction will be dismissed after a specified time-out.

Bounding Box Stabilization

The final step of the post processing is made with focus on the application area of the system, namely the assistance - and future automation - in operating a RWS. When presenting a bounding box for an object, the placement of this is equally important to the confidence score of the detection. In the video *RCBoeing747*, showing an airplane with a relatively smooth flying pattern, some fluctuations in the raw tracking prediction is noticeable. Considering a real world application, change in the size of a rigid object in camera video will almost only occur when the object moves towards or away from the camera. Because of this, it will be relevant to constrain rapid change in bounding box size to a certain degree. This is implemented, as stated in step 3, by considering a number of previous bounding boxes. A history/stabilization buffer stores the last 10 bounding boxes, which is an arbitrary value fitting this purpose. The buffer size, with a video capture rate of 30 FPS, translates to storing *one third of a second* of the past predicted bounding boxes. The frames are stored in a “first in, first out” (FIFO) manner, with the oldest frame popped out when a new frame is stored. A stabilization, or size filtration, is performed by taking the *median value* of all the bounding boxes in the history buffer. The original position of the raw track is combined with the median size for the final prediction. This can be compared to a simple band-pass filtration, limiting outliers of both relatively small and large bounding boxes.

5.2.4 Video Display

The visualization is also an important component of a computer vision application. For illustration purposes, a video overlay is implemented to present tracking results and system information at runtime (the period of which the tracking application is executed). The overlay, as it is illustrated in Figure 5.3, includes a labeled bounding box and status information in the header and footer of the video. The coordinates used to draw the bounding box, and the standard chosen for the system implementation, is on the form (x, y, w, h): x, y for the top left corner, and w, h for the width and height of the bounding box. Status information on the selected target for the video, as well as the confidence of the latest detection, is presented in the header. This score is interesting, not only because of the accuracy, but it will also indicate whether the predicted track may timeout (as described in subsection 5.2.3) if it drops below the selected threshold. The footer presents the current frame processing rate of both the detector and tracker, and will also indicate status “No track” if the prediction fails.

5.2.5 Command Line Interface

When developing an application like the one presented in this chapter, it is necessary to focus on a modular design. Not only in the meaning of being able to switch out components of a system, but also having the option to choose different system settings from the command line. This will increase the simplicity of system testing, and help with the adaption process to an embedded platform. As a command line application, the **tracking-app.py** is implemented with the ability to list the the different command line options supported, and this can be seen in Listing 5.3. The command line user interface supports several options: source and format of the input video, target information, settings for the detection module, and whether the video should be displayed or saved to a new output file.

Some of the system information is displayed in the video overlay (Figure 5.3), but even more detailed information is presented in the desktop terminal. The information printed at runtime can be seen in Listing 5.4, and matches the command line options for the tracking application. Additional status information is printed with a thematic prefix: [!] - warning, [c] - capture, [d] - detector, [i] - information and [t] - tracker.

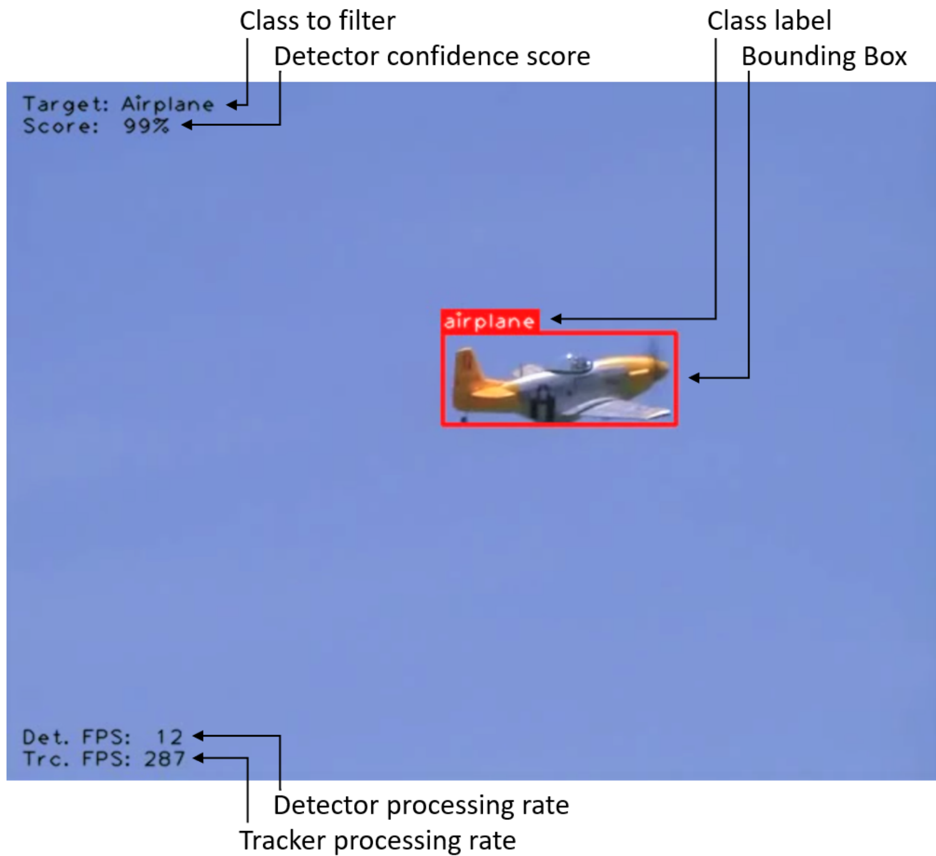


Figure 5.3: Frame from video displayed for the user, when tracking in video *HobbyKing*. The implemented overlay marks the object with a labeled bounding box in red, and provides additional relevant system information on top of the video.

```
$ python tracking-app.py -h
usage: tracking-app.py [-h] [-i SRC] [-t TARGET_CLASS]
                       [-th SCORE] [-s WIDTH HEIGHT]
                       [-f FPS] [-m MODEL_NAME]
                       [-l LABEL_NAME] [-w]
                       [-o FILE_NAME] [-c FOURCC]
                       [-e EXT]

optional arguments:
  -h, --help            show this help message and
                        exit
  -i SRC, --input SRC   path to video source
  -t TARGET_CLASS, --target TARGET_CLASS
                        target class to track
  -th SCORE, --threshold SCORE
                        detection score threshold
                        (0-100)
  -s WIDTH HEIGHT, --size WIDTH HEIGHT
                        video frame size
  -f FPS, --fps FPS     video playback rate
  -m MODEL_NAME, --model MODEL_NAME
                        name of inference model
  -l LABEL_NAME, --label LABEL_NAME
                        name of label file
  -w, --write           whether to write results
                        to file
  -o FILE_NAME, --output FILE_NAME
                        name of output file
                        (w/o ext)
  -c FOURCC, --codec FOURCC
                        fourcc codec for output
                        file
  -e EXT, --ext EXT     ext (container) for
                        output file
```

Listing 5.3: Application usage information displayed in the Ubuntu terminal.

```
$ python tracking-app.py
 ____  ____   __    ___  __ _   __    __ _   ___
(_  _)(  _ \ / _\  / __)(  / ) (  ) (  ( \ / __)
  )(   )   //    \( (__  )  (   )(  /    /( (_ \
 (__) (__\_)\_/\_/ \___)(__\_) (__) \_)__) \___/
tracking-app v1.0.0 (C) weedle1912

--- Source ---
* Input:     ../videos/HobbyKing.mp4
* Size:      640x480
* FPS:       30.0
--- Detector ---
* Model:     ssd_mobilenet_v2_coco_2018_03_29
* Labels:    mscoco_label_map
--- Object ---
* Target:    airplane
* Threshold: 50%

--- Running app:
[i] Init.
[d] Starting.
[c] Starting.
[i] Press "Esc" key to stop.
[d] Stopping.
[c] Stopping.
```

Listing 5.4: Application runtime information displayed in the Ubuntu terminal.

5.2.6 Data Processing Scripts

In addition to the implementation of the tracking application, it is necessary to initialize test data before testing the tracker, and calculate the resulting performance for evaluation. Thus, an effort is made to automate as many of these tasks as possible, both for this thesis and further development. The convention chosen for the application is to store a *normalized* bounding box by comma separated values (.csv) on the form "<x-pos>,<y-pos>,<width>,<height>". This is for a bounding box with top left corner at position (x, y), and side lengths *width* and *height*. The absence of a bounding box is stated with empty brackets "()". Each line in a csv-file for bounding boxes corresponds to the same frame in a video file, and this format is compatible with a spreadsheet for plotting of results. The following implemented scripts are located in **host/scripts** (see section A.3), and all of them support command line options to provide the necessary input arguments.

Data processing

- *images_to_video.py*:
  Create video from images in specified folder
- *normalize_gt.py*:
  Normalize ground truth file with rights to specified size
- *make_gt.py*:
  Make ground truth bounding box for object, frame by frame, in specified video
- *show_bboxes.py*:
  Play or step through video with specified bounding box file

The first two scripts in the list are aimed at formatting test data from benchmarks, because the data is stored as separate frames and the annotations are listed in terms of pixels by the original frame size. The third script is used to create custom annotations, as with the case of the *HobbyKing* video. The last script is used to visually review performance results.

Metrics

- *iou_by_csv.py*:
  Calculate the overlap score, according to Equation 3.5, for bounding boxes specified by two csv-files
- *dist_score_by_csv.py*:
  Calculate the distance score, according to Equation 3.8, for bounding boxes specified by two csv-files

5.3 Installation

5.3.1 Host

The host application is intended to run on an Ubuntu 16.04 compatible platform (only distribution tested), and the following instructions assume that the source code for the final implementation, *detection-and-tracking-v1.0.0*, is downloaded. The installation process of the tracking application is intended to be as smooth and straightforward as possible. By using a virtual environment, it is possible to create a self-contained directory tree, and structure the relevant additional software packages in a *requirements* file. In addition to this, there is a "setup" script implemented to automate most of the process. As it is instructed in the GitHub repository, which can also be seen in section A.2, there are only four simple steps to install the application:

1. Install the newest version of Protobuf (Google's language neutral buffers for serializing data)
2. Install virtual environment (*virtualenv*)
3. Run the *environment setup* script
4. Compile the Protobuf libraries

For the steps in 1, 2 and 4, the bash terminal commands are provided. When running the environment setup script in step 3, the virtual environment is created, the requirements are installed (including the OpenCV contribution package for the tracking API), and the TensorFlow object detection API is downloaded. The final step initializes the object detection API with the required Protobuf libraries.

The installation process is with this reduced to a more or less "copy-paste" process, as all the necessary commands are provided.

5.3.2 Target

For the target part of the system, a similar *environment setup* script is provided, and a few optimization notes for the Jetson TX2 platform. This can be considered as a starting point for future deployment on the embedded platform, and the details on this can be seen in section A.5. Because of the focus on the host application in this thesis, no further work was put into software for the Jetson TX2.

Part III

Evaluation

"An ounce of performance is worth pounds of promises."

Mae West

Chapter 6

Results

Contents:
This chapter presents the performance results of the implemented tracking application. It is important to notice that different data sets and videos are used to illustrate the different aspects of this specific combination of detection and tracking methods.

Resources:
The videos used for the evaluation of the tracking application are, in addition to the custom video *HobbyKing*, selected videos from the CVLab database [58] as described in subsection 4.2.1. Because of the assumption that classes with irregular motion patterns can replace drones for testing (section 1.3), the single target videos selected are of class *person*. All results are from running on Intel® Core™ i5-3470 quad-core CPU, with 640×480 video at 30.0 FPS.

The videos used for evaluation:

- Airplane: *HobbyKing*
- Person: *BlurBody*, *Dancer2*, *David3*, *Human2*, *Jump* and *Woman*

The tracking metrics used for this chapter:

- The bounding box overlap IoU/ϕ as defined in Equation 3.2/ Equation 3.5
- The proposed distance score S_δ defined in Equation 3.8
- The hybrid measurement S_T defined in Equation 3.10

6.1 Tracking in Test Videos

6.1.1 Video: HobbyKing

Figure 6.1: Frames from tracking in video *HobbyKing*.

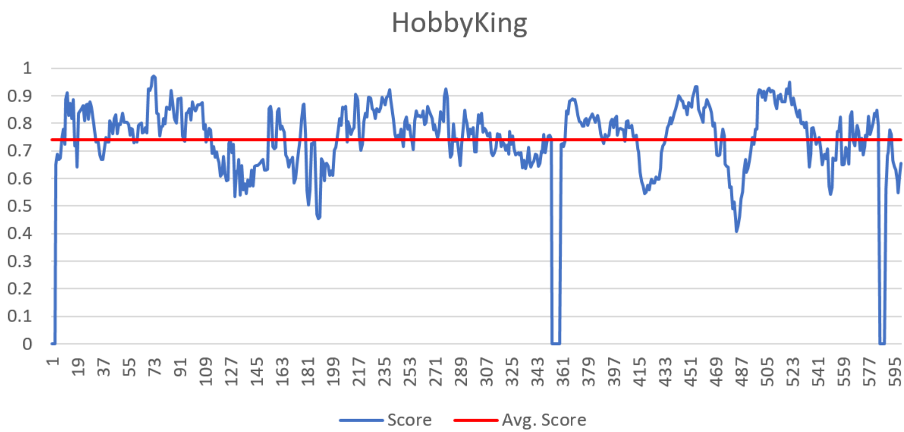


Figure 6.2: Plot of hybrid measure score S_T from tracking in video *HobbyKing*.

6.1.2 Video: BlurBody

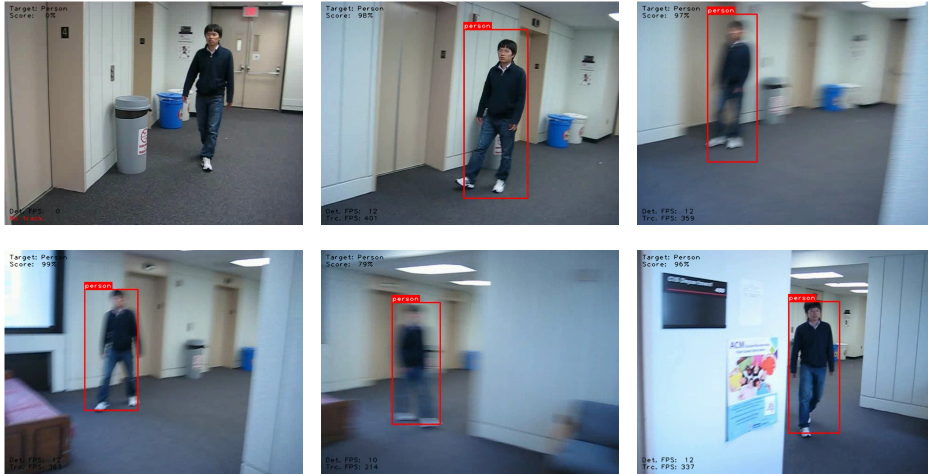


Figure 6.3: Frames from tracking in video *BlurBody*.

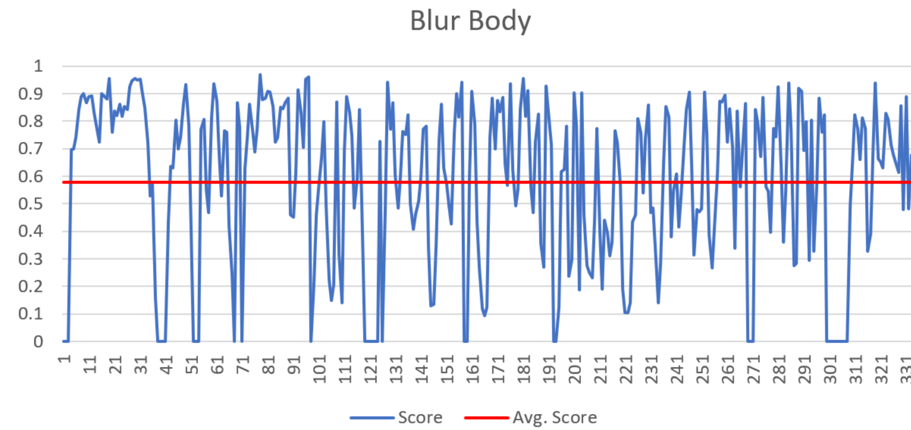


Figure 6.4: Plot of hybrid measure score S_T from tracking in video *BlurBody*.

6.1.3 Video: Dancer2

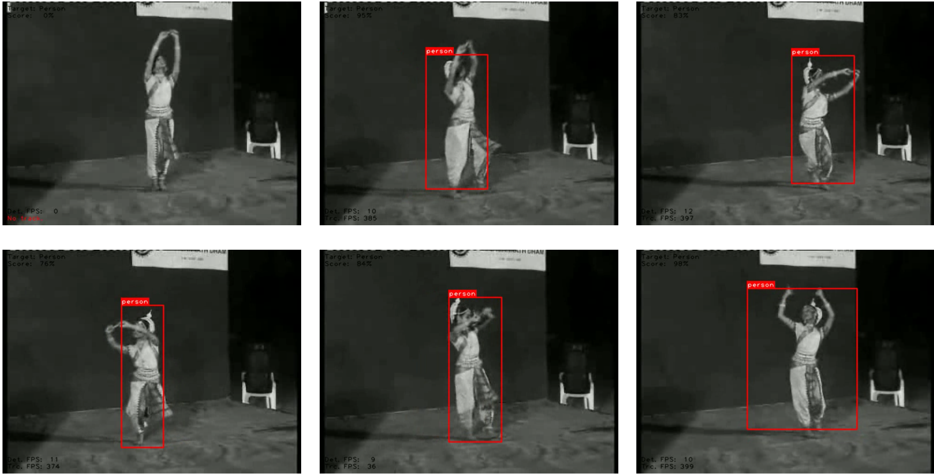


Figure 6.5: Frames from tracking in video *Dancer2*.

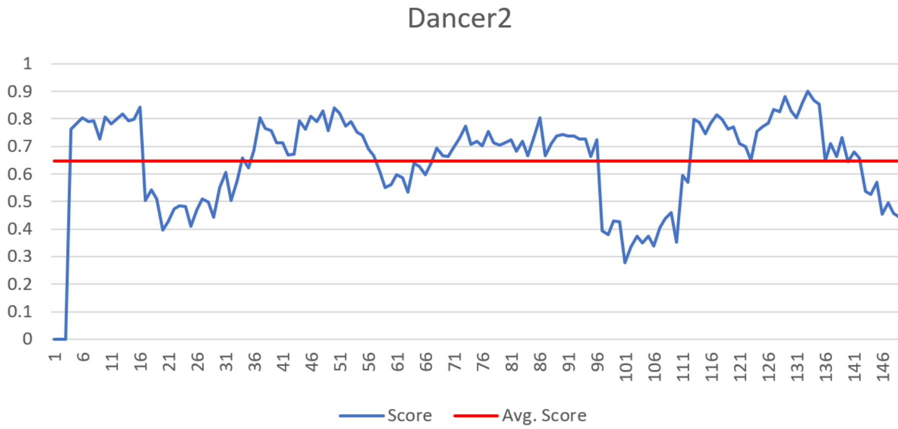


Figure 6.6: Plot of hybrid measure score S_T from tracking in video *Dancer2*.

6.1.4 Video: David3

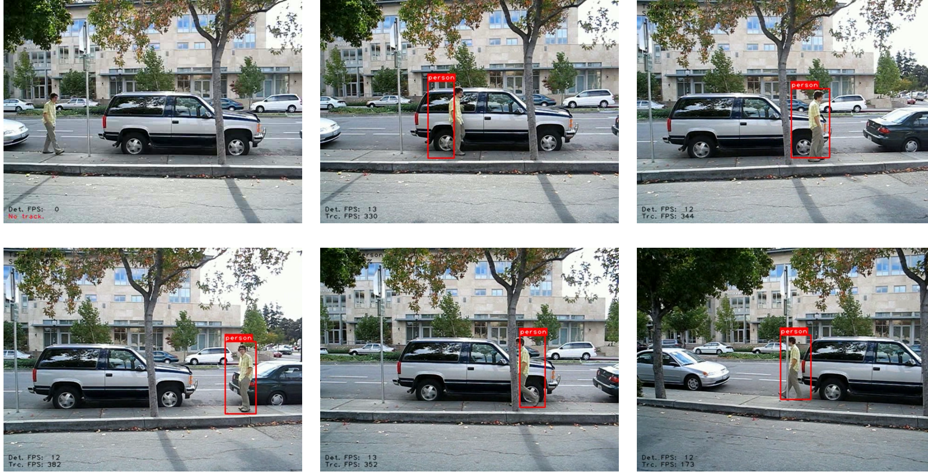


Figure 6.7: Frames from tracking in video *David3*.

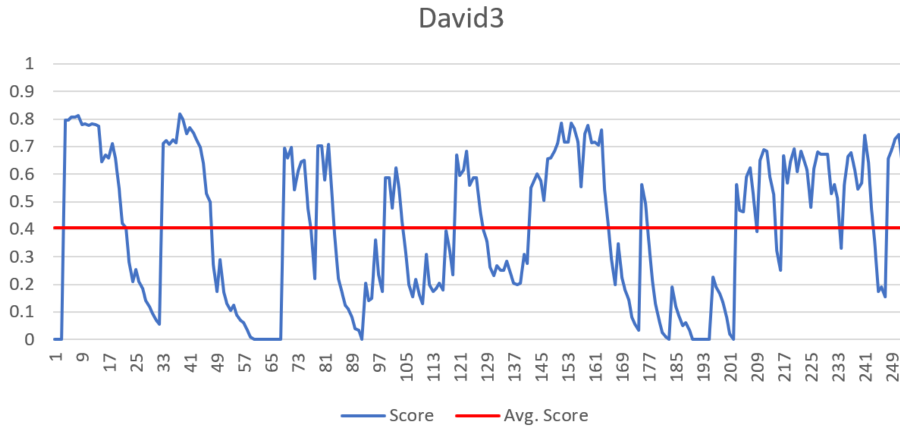


Figure 6.8: Plot of hybrid measure score S_T from tracking in video *David3*.

6.1.5 Video: Human2

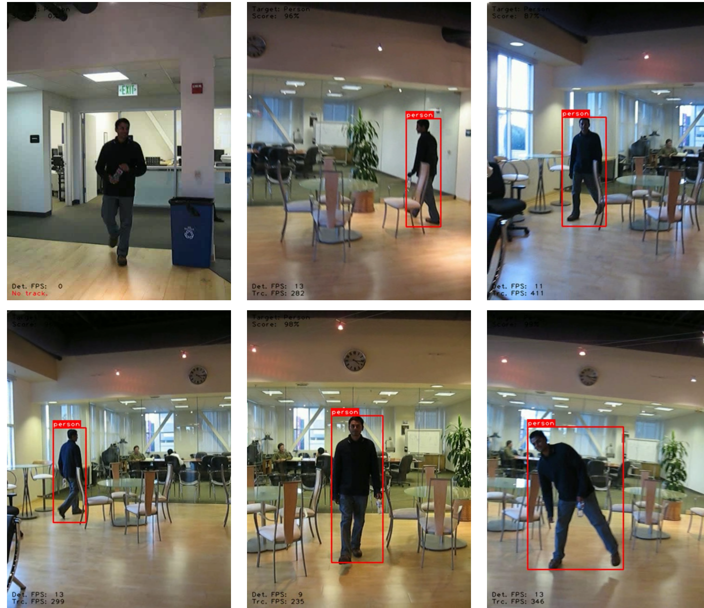


Figure 6.9: Frames from tracking in video *Human2*.

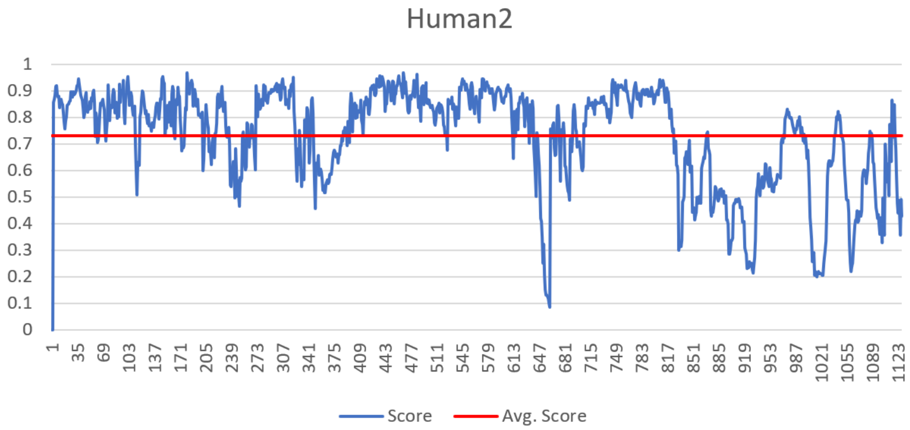


Figure 6.10: Plot of hybrid measure score S_T from tracking in video *Human2*.

6.1.6 Video: Jump

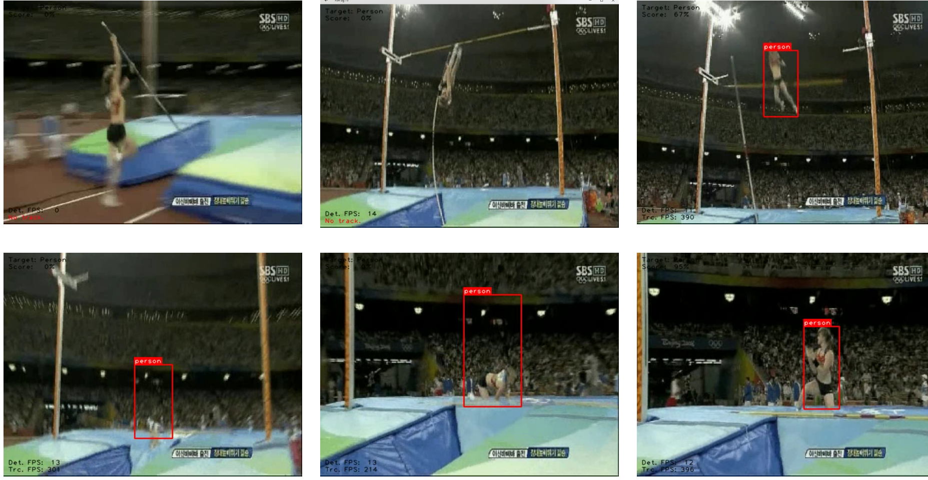


Figure 6.11: Frames from tracking in video *Jump*.

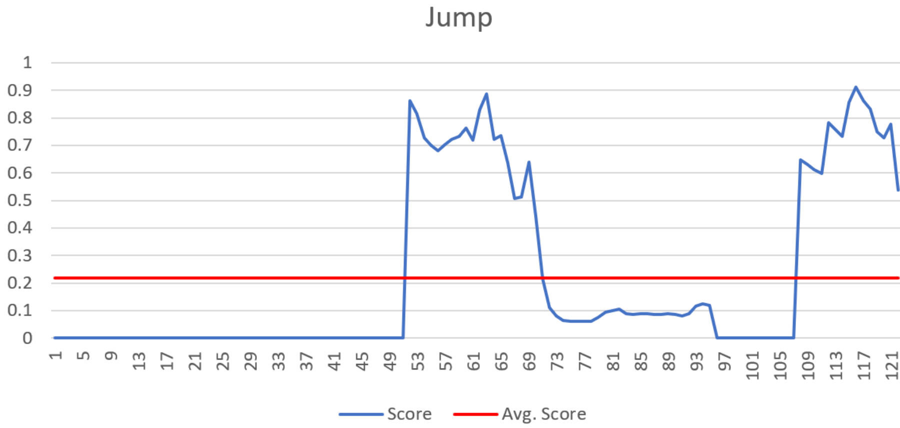


Figure 6.12: Plot of hybrid measure score S_T from tracking in video *Jump*.

6.1.7 Video: Woman

Figure 6.13: Frames from tracking in video *Woman*.

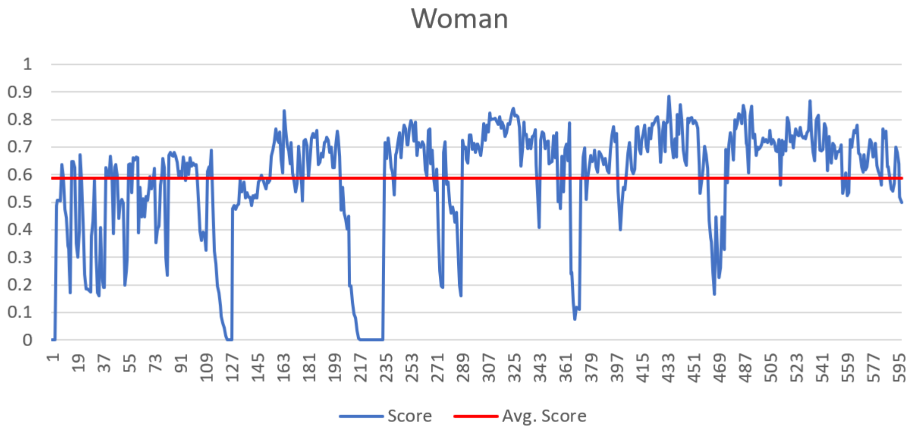


Figure 6.14: Plot of hybrid measure score S_T from tracking in video *Woman*.

6.1.8 Total Tracking Performance

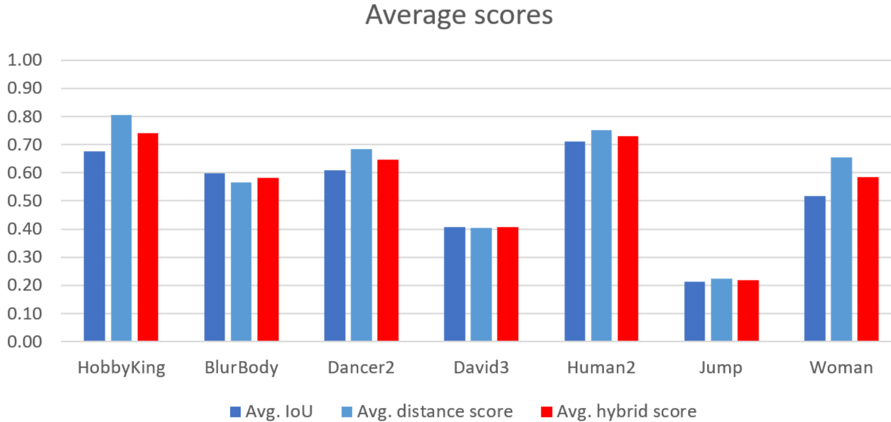


Figure 6.15: Average tracking scores for the test videos, the first entry is for class *airplane* and the rest for *person*. Both IoU and distance score S_d is combined in the hybrid tracking score S_T.

Table 6.1: Performance summary for tracking on test videos, presented as average scores. Detection with *ssd_mobilenet_v2_coco*, threshold 0.5 and tracker timeout 1.5s. The success rate is calculated for the hybrid score $S_{T,t} > \tau$.

Video	IoU ($\bar{\phi}$)	Dist. score ($\bar{S_\delta}$)	$\bar{S_T}$	Success rate ($\tau = 0.5$)
HobbyKing	0.68	0.81	0.74	0.97
BlurBody	0.60	0.56	0.58	0.64
Dancer2	0.61	0.68	0.65	0.80
David3	0.41	0.40	0.41	0.46
Human2	0.71	0.75	0.73	0.86
Jump	0.21	0.22	0.22	0.27
Woman	0.52	0.65	0.59	0.78

Chapter 7

Discussion

7.1 The Autonomous Tracker

7.1.1 Proof of Concept

The implemented tracking system, as it is described in chapter 5, can be considered a proof of concept for the objective of the thesis. In the problem description in section 1.2, the only directive provided is that the autonomous functionality should be explored as a combination of an accurate detector module with a fast tracking module. With the implementation of the Python application **tracking-app.py**, a solution to this combination problem is presented.

The detection module is implemented with the TensorFlow object detection API, and use a deep neural network based on a Single Shot Detector (SSD) architecture, which is considered state-of-the-art. The results presented in chapter 6 were produced using the *ssd_mobilenet_v2_coco* model, which is one of the modules presented in Table 4.6. With regards to the mAP scores, this was the model with the highest score that could comply with the design requirement defined in Equation 5.3. This was determined by testing the application with the different models listed, and choose the one with the highest mAP that did not provide a system warning (the message printed for the user, described in subsection 5.2.2). This can be a way to tune any other system configuration with an optimal detection model.

The tracking application is tested on an Intel® Core™ i5-3470 *CPU*. This means that even though the hardware configuration is with limited processing capabil-

ities, the application can still meet the real-time requirements of live camera video.

7.1.2 The Embedded Platform

The system implementation for the autonomous tracker is made with focus on the Nvidia Jetson TX2 module. This is done by choosing a framework which is known to be compatible with the embedded platform. Google's TensorFlow provides deep learning models which are supported by a number of relevant features, such as cloud computing, training visualization tools, and optimization with TensorRT. This is important for developing and customizing an effective detection module for future deployment. As for the tracking module, the API used for the tracking algorithms are provided by OpenCV. This is a practical approach, because a custom distribution of OpenCV is part of the software bundle for the Jetson TX2 (JetPack 3.2). The decisions on third-party software are made with focus on making the step towards embedded deployment easier.

It should also be mentioned that applying state-of-the-art methods to solve a practical problem will lead to a few issues. The Jetson TX2 platform does indeed support a custom Ubuntu distribution, but such embedded hardware comes with limitations. Considering that most of the third-party software used for the application is under development, chances are that these might be incompatible with the Jetson TX2 at the time of writing. This could for instance be the case with the tracking API, as it is not part of any official OpenCV release - and thus it will not be part of the custom version for the Jetson TX2. However, precautions for such issues are made when designing a module based architecture, such as the one presented in this thesis. Because of this, it is possible to make subtle adjustments - and even change out modules - without breaking the tracking application.

7.1.3 Limitations

The steps taken to process the raw detection and tracking predictions are presented in subsection 5.2.3, but as mentioned in that section, they are not without room for improvement.

The most important consideration should be made with regards to how multiple detections are reduced to one, for the purpose of single target tracking. As the application stands now, with the greedy approach of choosing the best detection, it is not fit for the working environment of the RWS. The problem with this is that it can not handle a scene of multiple targets of interest, as the confidence among several instances of the same class may change - and thus the tracker will jump between these sporadically. Additional requirements for this selection

should be considered, by for instance choosing the detection that is closest to the previous prediction. This data association problem can be solved for several levels of complexity, but at least a few more such requirements should be considered.

When it comes to the tracker, the downside of the OpenCV API chosen is that the only interface to the tracker is *initialization* and *update*. An ideal implementation should have the opportunity to adjust the tracker, not only create and start a new one. This simple interface limits the control of the tracker, and the tracking state history can be difficult to consider. It can be argued that the use of a frame buffer is a way of compensating for this, as it is possible to store some of the past tracking information. If additional functionality and control was provided, it could be possible to have a more nuanced selection of the final prediction and even determine the confidence of the tracker as well. Instead of blindly using a new detection, the current track could be evaluated and just kept going.

7.2 Performance in Test Videos

7.2.1 The Selected Videos

Tracking of *drones* is probably one of the most interesting areas of research for future RWS vision based systems, both because of the increasing safety threats and the issues related to object size and movement. Because of the lack of relevant annotated drone data, other object classes are used to evaluate the performance of the implemented tracking application. The class *airplane* is a close alternative - if not even a parent class - for drones, with a similar operation environment. For argument's sake, the Oxford English Dictionary defines "drone" as: "*A remote-controlled pilotless aircraft or missile.*". This means that the custom videos of remote-controlled model airplanes does fit the definition (subsection 4.2.2). So-called *fixed wing drones*, which can impose a real world threat, does also have a resemblance to small airplanes. Nevertheless, for the environment of the RWS and the defence perspective, it is implied that the drone is either of military grade or a multicopter.

As for the videos used from the CVLab data set [58], they all depict the class *person*. None of the videos in the additional data sets are of airplanes, and so this second class have to compensate for this. The use of *person* can be justified because it is one of the original classes proposed in the preliminary research [1] (see subsection 1.1.3), and the class is included in the Microsoft COCO data set [56]. This is important because the relevant neural network models listed in Table 4.6 are all pre-trained on this data set, which means that this test class will be compatible with the modules of the system.

What is common for all the selected videos, is that they all include only a *single instance* of the relevant object class, due to the focus on single target tracking. This is the only criteria used for selection of videos from the CVLab data set, to be able to present an objective evaluation of the tracking performance. This is not completely the case for the custom videos, as these are subjectively made to mimic relevant test cases for drones. However, the video *HobbyKing* was manually annotated for evaluation purposes, because it is the one video with the most relevant scenarios (blur, size change, rotation and erratic movement).

7.2.2 Evaluation of the Results

The tracking performance is only plotted as the hybrid measure S_T at each frame for all of the videos. It is therefore interesting to first consider the performance summary in either Figure 6.15 or Table 6.1, to see how both the overlap scores and distance scores contribute to these results. For a majority of the videos, the

distance score prove to be higher than the overlap score, which can indicate that the tracking application is better at predicting the position of the object. This difference is clearly the case for both *HobbyKing* in Figure 6.2 and *Woman* in Figure 6.14. A possible explanation for this could be how the final prediction is stabilized with rights to size, as described in subsection 5.2.3. Both of these videos are subject to relatively rapid change in size of the object, as the camera is zoomed in on the objects. The stabilization process is a contributing factor for the presented bounding box size, and is intended to limit sudden change. This will affect the response time when the object really does change at a rapid rate. This does not have to be a persistent problem, as the stabilization process can be further tuned for the application area.

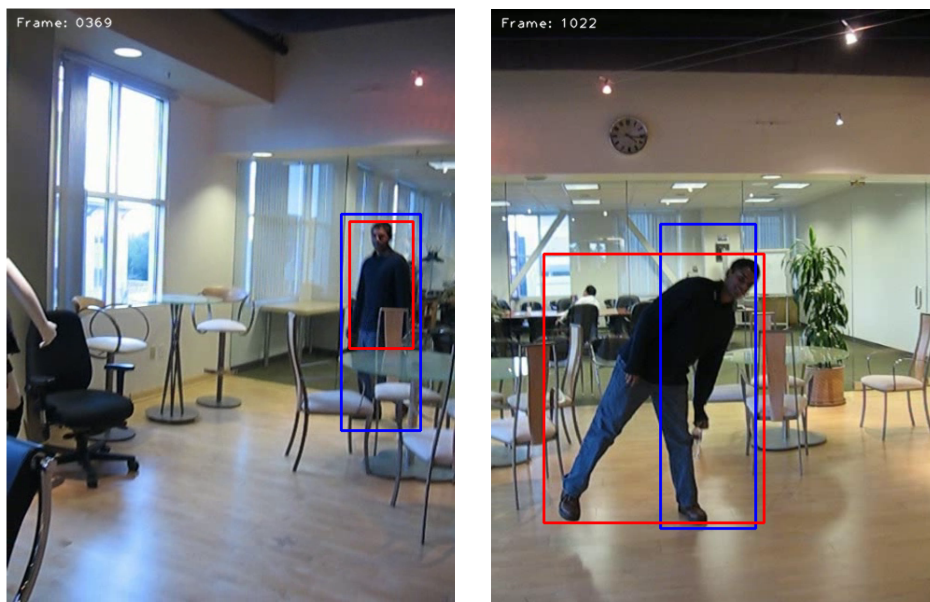
For the two worst performances, *David3* in Figure 6.8 and *Jump* in Figure 6.12, it is not as easy to state what the problem might be. For *David3*, the plot illustrates in "waves" how the prediction periodically drifts away from the target (or how it seems like the person is walking away from a stationary bounding box). The feature points in the background may be more distinct than those of the object at each detection, and some occlusion (a pole and a tree) disrupts the object flow - which can be the causes of the drift. For the *Jump* video, the object seems too blurry and distorted in the image, and can not be detected until half way through. Again, this is also an example of how the size of the bounding box can not keep up with the change, which can be seen in the still images in Figure 6.11.

The score plot for *BlurBody* in Figure 6.4 is a good correlation of what the video depicts. With the continuous shake of the camera in the horizontal plane, it is noticeable from Table 6.1 that the average distance score S_δ weakens the overall performance. The final two videos, *Dancer2* in Figure 6.6 and *Human2* in Figure 6.10, both show relatively stable results. The "dips" in the score plot for both of these videos correlate with change in posture, and distortion of the object. This issue is not necessarily a weakness of the implemented tracking application, and is further discussed in subsection 7.2.3.

Compared to the summary chart in Figure 6.15, an additional measure is added to the summary in Table 6.1. A success rate is usually considered according to the overlap score [27], with a threshold $\tau = 0.5$. But in this table, the success rate is calculated according to the hybrid measure S_T because this is the score used for all the plots. Even though it can be difficult to evaluate such a rate, it should count positively for the performance that many of the test videos show an approximate 0.80 success rate - especially with the airplane video at *0.97*.

7.2.3 Bounding Box Consideration

Both the definition of a bounding box, and what a "best fit" is, may be dependent on the CV application. It is important to consider the performance results with reference to the video annotations. For instance, the plot in Figure 6.10 (for the video *Human2*) shows good tracking performance, and it would be fair to say that a few mistakes are expected. Two frames from this video, at two of these "dips" in the score plot, is shown in Figure 7.1. These frames indicate that there have been a different approach to what is considered a good ground truth annotation, compared to the bounding box predicted by the tracker. In the case of occlusion (Figure 7.1a), the tracker is predicting the visible part of the object, and for movement (Figure 7.1b) the tracker predicts a bounding box to fit the object - and does not keep the original pose as the annotation. It can be argued that a bounding box should make a close fit around the object, especially when the size of the object is important for the tracking scenario.



(a) Difference in object occlusion.

(b) Difference in non-rigid movement.

Figure 7.1: The difference in bounding box approach affects the performance results. The blue box is the ground truth provided for the video, and the red box is the output of the tracking application.

7.2.4 The Proposed Hybrid Measure

Without any means of comparing this measure with other trackers, the hybrid performance score S_T (Equation 3.10) is still used for the evaluation. For all of the seven videos presented in chapter 6, this score is plotted to visualize the tracking performance. This is because it is considered to represent a more intuitive approach to tracking assessment, with a positive focus on both the overlap and the distance score.

With the discussion in subsection 3.3.2 of different approaches for tracker assessments, it is stated that there is no “de-facto” standard [27]. This can be considered as an opportunity to present any measure that seems fit for the application. For the case of the RWS, the size and placement of a bounding box will translate into important target characteristics. This focus seems to be common for other applications as well, because the *overlap* and the *distance error* are considered to be basic measures. The problem of leaving one of these out of the evaluation is that they really are closely related. It can be difficult to present only the overlap score, because this value does not say anything about what part of the object that is covered. For the case of a sole distance error measure, the position can be evaluated as good even if the size is completely off.

A previous attempt at a hybrid measure is in subsection 3.3.3 considered to be “[...] two very different basic measures [...] combined in a rather complicated manner, prohibiting a straightforward interpretation”. By assuming that this is because the distance error is measured in pixels, the new distance score S_δ is proposed to deal with the problem. The abstraction of comparing the ground truth box to a shooting-target completely disregards any pixel measure or frame scale, and considers only the edges of the bounding box. This new approach on the distance error, turning it into a score rather than a “penalty”, creates an environment of which these two measures can be combined. A hybrid measure like this also highlights how both size and position are important to predict a good bounding box.

With the definition of the hybrid measure in Equation 3.10, the intention is to rule the overlap and distance score equally important. The effect of this can be further explored when considering the performance plots in chapter 6. Intuitively it is still possible to have a positive overlap score when the distance score is zero, as will be the case of most of the smaller values plotted. With an average value, the overlap score is in such a case reduced by half - but it is still a small contributing value. Thus, it may be suitable to use a constraint on the hybrid score S_T, with a more strict consideration of the distance score. One of the reasons for introducing the distance score S_δ in the first place, is to determine any prediction

outside of the object bounds as "off-target". This abstraction can be extended to the hybrid tracking measure with a condition as presented in Equation 7.1.

$$S_{T,t} = \begin{cases} \frac{1}{2}(S_{\delta,t} + \phi_t), & \text{if } S_{\delta,t} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

With this condition ($S_{\delta,t}$ defined in Equation 3.8) it will be possible to determine when the tracker has lost the target, without using any arbitrary or application specific threshold.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

A step towards autonomous functionality for a Remote Weapon Station can be made by processing live camera video with state-of-the-art computer vision methods. The autonomous tracker proposed in this thesis is a proof of concept for the combination of a detection module and a tracking module. For future deployment on the Jetson TX2 embedded platform, it is imperative that any third-party software utilized supports versatile use of supervised learning and runtime optimization. Even though the use of TensorFlow and OpenCV is suggested, the module-based architecture is intended to provide interchangeability. However, the solution presented creates a finite limit on how accurate the detection module can afford to be, as the processing rate of the two modules are inversely proportionate. This must be considered when editing any future application configurations.

The performance of the final tracking application is presented as a hybrid measure, which is a combination of both the bounding box overlap and a new proposed distance score. This measure is intended to present an intuitive evaluation of the tracker performance. With satisfactory results on several videos with relevant object behavior, and the proven compliance with the real-time restriction on live camera video, it can be concluded that this approach on autonomous tracking is compatible with the use case of a Remote Weapon Station.

8.2 Further Work

8.2.1 Work-flow

First of all, a continued effort should be put into deployment on the Jetson TX2. The tracking application should be edited to run on the embedded platform. This should be done with focus on the intended work-flow of TensorRT (Figure 4.1, subsection 4.1.1), by dividing the system into a *host* and a *target* partition. A neural network model is serialized on a host computer, and further optimized and deployed on the target platform. With the suggested framework TensorFlow, this will translate into using a desktop computer for supervised learning and model configuration, and only transfer an optimized detection model to the Jetson TX2.

8.2.2 Implementation

The current state of the tracking application is intended to create an easy transition for further work, and the GitHub repository [9] is a contribution for this purpose. With an immediate continuation of this work, two things should be considered:

- The use of *GStreamer*

- Further use of OpenCV

As it is listed in the beginning of chapter 4, GStreamer is included in the JetPack 3.2 software bundle. This is a library for media handling. With GStreamer, it is possible to construct *pipes* optimized for the Jetson TX2 to encode and decode video efficiently - to minimize video latency. The video capture module implemented for the tracking application uses OpenCV to capture video, and OpenCV can be built with GStreamer support. This means that instead of providing a video file, the input source for the module will be a video pipe.

Further use of OpenCV must be considered because the custom version provided for the Jetson TX2 may not be adequate. Additional support is required to use the tracking API, and potentially GStreamer, which will be on the cost of the optimized version provided for the platform. This does not, however, need to be the only approach. With the implementation of class-oriented detection and tracking modules, it is possible to update and/or change the functionality of the application through the respective modules. As OpenCV is a library for computer vision methods, it is possible to implement a custom tracker better suited for this purpose. This can also be an approach for dealing with the interface limitations discussed in subsection 7.1.3, and have more control of the tracking algorithm.

8.2.3 Data Set

The amount of available annotated data will always be a concern when relying on supervised learning. A proof of concept with a different object class than desired will only reach as far as to an initial phase of the development process. At a certain point, relevant training data must be provided. The problem of scarce annotated data can be solved by different approaches, both with real images and artificial [12]. Either way, a protocol for *how* the data is annotated must be considered, to be able to avoid issues as the one discussed in subsection 7.2.3. An ideal ground truth annotation will reflect how the bounding box is supposed to be predicted.

Appendix A

The GitHub Repository

The *detection-and-tracking* repository, created for this thesis, is publicly available at GitHub [9].

A key intention with this thesis is to provide a solid foundation for further development of a tracking system on an embedded target.

The application software is structured by the following hierarchy:

```
detection-and-tracking/
|--- host/
     |--- scripts/
     |--- src/
          |--- detector/
          |--- tracker/
          |--- utils/
     |--- test/
          |--- cvlab/
          |--- vot2017/
|--- target/
     |--- scripts/
|--- videos/
     |--- data/
```

A.1 detection-and-tracking/

weedle1912 / **detection-and-tracking**

Autonomous object tracking: A combination of a detector and a tracker

#object-detection #object-tracking #jetson-tx2 #tensorflow-experiments #opencv-python

182 commits	2 branches	1 release	1 contributor	MIT

Branch: master ▾ | New pull request | Find file | Clone or download ▾

weedle1912 Merge pull request #15 from weedle1912/develop ... Latest commit f4b0538 2 days ago

host	Deleted conflicting README	2 days ago
target	OpenCV installation guide	a month ago
videos	Finished gt for HobbyKing	18 days ago
.gitignore	Add empty object_detection folder as placeholder	2 days ago
LICENSE	Added MIT license	2 months ago
README.md	Added links to required API's	2 days ago

README.md

Detection and Tracking

Repository for Master's Thesis in Engineering Cybernetics at NTNU, 2018.

Title: *Autonomous Target Detection and Tracking for Remotely operated Weapon Stations*

Intention

Detect and track targets of interest in camera video, for a Remote Weapon Station (RWS).

Approach

Combine an accurate detector with a fast tracker. Methods of interest:

- Detector based on *deep learning*
- *Point based* tracker

Motivation/Design

If a detector process frames slower than the video frame rate, a number of frames will be skipped in between detections - and the detection will also be a few frames "old" when presented. A solution to this is to buffer the skipped frames, and retrace this buffer with a fast enough tracker, to make the detection relevant for the current frame.

The implementation `host/tracking-app.py` is an *autonomous tracker* - a real-time tracker with periodic corrections from a deep learning detector.

Software Framework

- TensorFlow: Google's TensorFlow Object Detection API
- OpenCV: OpenCV-contrib-python Object tracking API

Hardware

- Host: Desktop PC (optional CPU/GPU depending on TensorFlow distribution)
- Target: Nvidia Jetson TX (future work)

A.2 host/

weedle1912 / detection-and-tracking

Branch: master ▾ detection-and-tracking / host / Create new file Find file History

weedle1912 Deleted conflicting README Latest commit 93580e2 2 days ago

..		
scripts	Created README	2 days ago
src	Deleted conflicting README	2 days ago
test	dir 'host/val' => 'host/test'	2 days ago
README.md	Edit header size	2 days ago
requirements.txt	Update README with instruction, and added env_setup script	2 days ago
tracking-app.py	Update print	2 days ago

README.md

Host: PC with Linux

1. Description

Purpose:

Object tracking application running on host computer.

Requirements:

- Ubuntu 16.04
- TensorFlow 1.8
- Protobuf 3.0
- Google's object_detection API
- OpenCV-contrib for tracking API

2. Setup

2.1 Install the newest version of Protobuf

```
$ curl -OL https://github.com/google/protobuf/releases/download/v3.2.0/protoc-3.2.0-linux-x86_64.zip
$ unzip protoc-3.2.0-linux-x86_64.zip -d protoc3
$ sudo mv protoc3/bin/* /usr/local/bin/
$ sudo mv protoc3/include/* /usr/local/include/
$ rm -r protoc*
```

2.2 Install virtual environment

```
$ sudo apt-get install -y python-pip python-dev python-virtualenv
```

2.3 Run the setup script

```
$ chmod +x ./scripts/env_setup.sh
$ ./scripts/env_setup.sh
```

2.4 Compile Protobuf libraries

```
$ cd src/detector
$ sudo protoc object_detection/protos/*.proto --python_out=.
$ cd ../..
```

3. Run

Activate the virtual environment, and run application:

```
$ source ~/tensorflow/bin/activate
(tensorflow)$ python tracking-app.py
```

A.3 host/scripts/

weedle1912 / **detection-and-tracking**

Branch: master ▾ **detection-and-tracking** / host / **scripts** /

Create new file | Find file | History

weedle1912 Created README — Latest commit eb228f4 2 days ago

..		
README.md	Created README	2 days ago
dist_score_by_csv.py	Avg. calculated from all values	2 days ago
env_setup.sh	Update README with instruction, and added env_setup script	2 days ago
images_to_video.py	Script to make video of image files	24 days ago
iou_by_csv.py	Avg. calculated from all values	2 days ago
make_gt.py	Changed name of scripts	2 days ago
normalize_gt.py	Changed name of scripts	2 days ago
show_bboxes.py	Changed name of scripts	2 days ago

README.md

Collection of practical scripts for testing and development

Setup

- `env_setup.sh` :
  - Create virtual environment ~/tensorflow
  - Install requirements
  - Download TF Object Detection API

Data processing

- `images_to_video.py` :
  - Create video from images in specified folder
- `make_gt.py` :
  - Make ground truth bbox for object, frame by frame, in specified video
- `normalize_gt.py` :
  - Normalize ground truth file with rights to specified size

- `show_bboxes.py` :
  - Play or step through video with specified bbox file

Metrics

- `iou_by_csv.py` :
  - Calculate the "intersection over union"/overlap for bboxes specified by two .csv-files
- `dist_score_by_csv.py` :
  - Calculate a distence score for bboxes specified by two .csv-files
  - Measure of prediction center relative to ground truth center: 1 for overlap to 0 at ground truth edge
  - Formula: 1 - min(max(2 * error_x_dir / w_G, 2 * error_y_dir / w_G), 1)

A.4 host/test/

weedle1912 / detection-and-tracking

Branch: master ▾ detection-and-tracking / host / test / Create new file Find file History

weedle1912 dir 'host/val' => 'host/test' Latest commit a96bacb 2 days ago

..

cvlab	dir 'host/val' => 'host/test'	2 days ago
vot2017	dir 'host/val' => 'host/test'	2 days ago
README.md	dir 'host/val' => 'host/test'	2 days ago

README.md

Setup

The following scripts can be used to initialize this directory with the relevant videos (as specified in [cvlab, vot2017]/list.txt):
CVLab:

```
$ chmod +x cvlab/cvlab_setup.sh
$ cd cvlab
$ ./cvlab_setup.sh
```

VOT2017:

```
$ chmod +x vot2017_setup.sh
$ cd vot2017
$ ./vot2017_setup.sh
```

Videos

Name	Source	Object class
BlurBody	cvlab	Person
Dancer2	cvlab	Person
David3	cvlab	Person
Human2	cvlab	Person
Jump	cvlab	Person
Woman	cvlab	Person
iceskater1	vot2017	Person
graduate	vot2017	Person

A.5 target/

weedle1912 / detection-and-tracking

Branch: master ▾ detection-and-tracking / target / Create new file Find file History

weedle1912 OpenCV installation guide Latest commit f25fc66 on 26 Apr

..

scripts	Script: OpenCV gen Makefile	a month ago
README.md	OpenCV installation guide	a month ago

README.md

Target: NVIDIA Jetson TX2 Development Kit

1. Description

Purpose:

System deployment.

Requirements:

(These should be installed when flashing the TX2 with JetPack 3.2)

- CUDA Toolkit 9.0
- cuDNN 7.0
- TensorRT 3.0
- OpenCV 3.3.1

2. Setup

2.1 Flash TX2 with JetPack 3.2

Include the following installations:

- CUDA Toolkit 90
- cuDNN 7.0
- TensorRT 3.0
- OpenCV 3.3.1

2.2 Build and install OpenCV from source

1. Remove old OpenCV installations and install necessary dependencies:
   `$ ./scripts/OpenCV_prep.sh`
2. Edit the file `/usr/local/cuda/include/cuda_gl_interop.h` to patch OpenGL related compilation problems. This is done by commenting out the lines #62-66 and #68. Lines #62-68 should look like:

```
//#if defined(__arm__) || defined(__aarch64__)
//#ifndef GL_VERSION
//#error Please include the appropriate gl headers before including cuda_gl_interop.h
//#endif
//#else
#include <GL/gl.h>
//#endif
```

And fix symbolic link:

```
$ cd /usr/lib/aarch64-linux-gnu/
$ sudo ln -sf tegra/libGL.so libGL.so
```

3. Download source and create Makefile for OpenCV: `$ ./scripts/OpenCV_init_build.sh`

3. Optimize performance on the TX2

3.1 Max frequency to CPU, GPU and EMC clocks

Maximize jetson performance by setting static max frequency to CPU, GPU and EMC clocks, with the `~/jetson_clocks.sh` script:

1. Set max:
   1.1 Store current settings ([file] is optional, default is `{$HOME}/l4t_dfs.conf`):
   `$ ~/jetson_clocks.sh --store [file]`
   1.2 Set static max frequency:
   `$ ~/jetson_clocks.sh`
2. Restore previously stored settings:
   `$ ~/jetson_clocks.sh --restore [file]`

3.2 Specify different CPU and GPU modes

With the command line tool `nvpmodel`, the TX2 can use different power modes for CPU and GPU frequencies, as well as CPU core activation: `$ sudo nvpmodel -m [mode]`

The following models are predefined (quad core ARM A57 and dual core Denver):

Mode	Mode name	Denver	Frequency	Arm A57	Frequency	GPU Frequency
0	Max-N	2	2.0 GHz	4	2.0 GHz	1.30 Ghz
1	Max-Q	0		4	1.2 Ghz	0.85 Ghz
2	Max-P Core-All	2	1.4 GHz	4	1.4 GHz	1.12 Ghz
3	Max-P ARM	0		4	2.0 GHz	1.12 Ghz
4	Max-P Denver	2	2.0 GHz	0		1.12 Ghz

These modes are defined in `/etc/nvpmodel.conf`, which can be updated with custom mode definitions.
Check current mode with:
`$ sudo nvpmodel -q --verbose`

A.6 videos/

weedle1912 / detection-and-tracking

Branch: master ▾ detection-and-tracking / videos / Create new file | Find file | History

weedle1912 Finished gt for HobbyKing Latest commit c1248bc 18 days ago

..		
data	Finished gt for HobbyKing	18 days ago
BlueAngels.mp4	Added test video	2 months ago
HobbyKing.mp4	Added new test video	2 months ago
RCBoeing747.mp4	Added two test videos	a month ago
README.md	Added two test videos	a month ago
ToyPlane.mp4	Added two test videos	a month ago

README.md

Videos Origin

File	Source	Time	Objects
BlueAngels.mp4	*2017 Blue Angels NAF El Centro Air Show*	05:27-05:54	Airplane
HobbyKing.mp4	*Hobby King P-51D 1.2m with Movable Canopy and Retracts*	00:54-01:14	Airplane
RCBoeing747.mp4	*NEW BIGGEST RC AIRPLANE IN THE WORLD BOEING 747-400 VIRGIN ATLANTIC AIRLINER*	01:15-02:45	Airplane
ToyPlane.mp4	*WLToys Cessna 182 RC Plane Unboxing, Build, Review, and Maiden Flight*	06:06-07:47	Airplane

Bibliography

[1] Vetle Bjørngaard Gundersen. "Autonomous Target Detection and Tracking for Remotely operated Weapon Stations". Preliminary Project. Norwegian University of Science and Technology (NTNU). Dec. 2017.

[2] Margaret A. Boden. *Mind As Machine: A History of Cognitive Science.* Oxford University Press, 2006.

[3] The New York Times. *A Drone, Too Small for Radar to Detect, Rattles the White House.* 2015. URL: `https://www.nytimes.com/2015/01/27/us/white-house-drone.html` (visited on 01/17/2018).

[4] New Scientist. *A swarm of home made drones has bombed a Russian airbase.* 2017. URL: `https://www.newscientist.com/article/2158289-a-swarm-of-home-made-drones-has-bombed-a-russian-airbase/` (visited on 01/17/2018).

[5] Kongsberg Defence & Airspace. *Remote Weapon Stations.* 2017. URL: `https://www.kongsberg.com/en/kds/products/remoteweaponstation/` (visited on 01/15/2018).

[6] R. Verschae and J. Ruiz-del-Solar. "Object Detection: Current and Future Directions". In: *Front. Robot. AI* 2.29 (2015).

[7] J. Joshan Athanesious and P. Suresh. "Systematic Survey on Object Tracking Methods in Video". In: *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)* 1.8 (2012), pp. 242–247.

[8] Advanced Television Systems Comitee Inc. *Video System Characteristics of AVC in the ATSC Digital Television System.* Document A/72 Part 1. ATSC, May 2015.

[9] Vetle Bjørngaard Gundersen (user: weedle1912). *Repo.: detection-and-tracking.* `https://github.com/weedle1912/detection-and-tracking`. 2018.

[10] A. F. Hurtado et al. “Proposal of a Computer Vision System to Detect and Track Vehicles in Real Time Using an Embedded Platform Enabled with a Graphical Processing Unit”. In: *2015 International Conference on Mechatronics, Electronics and Automotive Engineering (ICMEAE)*. Nov. 2015, pp. 76–80. DOI: 10.1109/ICMEAE.2015.24.

[11] C. Li, Y. Xi, and S. Ding. “Implement tracking algorithm using CNNs”. In: *2016 35th Chinese Control Conference (CCC)*. July 2016, pp. 7137–7141. DOI: 10.1109/ChiCC.2016.7554485.

[12] Cemal Aker and Sinan Kalkan. “Using Deep Networks for Drone Detection”. In: *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. Aug. 2017, pp. 1–6. DOI: 10.1109/AVSS.2017.8078539.

[13] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Pearson, 2012.

[14] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[15] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, 2010.

[16] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2016.

[17] N. Dalal and B. Triggs. “Histograms of oriented gradients for human detection”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*. Vol. 1. 2005, pp. 886–893.

[18] Balázs Kégl. “The return of AdaBoost.MH: multi-class Hamming trees”. In: *CoRR* abs/1312.6086 (2013).

[19] Zehang Sun, G. Bebis, and R. Miller. “On-road vehicle detection: a review”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.5 (2006), pp. 694–711.

[20] Jing Li and Nigel M. Allison. “A comprehensive review of current local features for computer vision”. In: *Neurocomputing* 71.10 (2008). Neurocomputing for Vision Research Advances in Blind Signal Processing, pp. 1771–178.

[21] Yali Li et al. “Feature representation for statistical-learning-based object detection: A review”. In: *Pattern Recognition* 48.11 (2015), pp. 3542–3559.

[22] S. R. Balaji and S. Karthikeyan. “A survey on moving object tracking using image processing”. In: *2017 11th International Conference on Intelligent Systems and Control (ISCO)*. 2017, pp. 469–474.

[23] Himani S. Parekh, Darshak G. Thakore, and Udesang K. Jaliya. “A Survey on Object Detection and Tracking Methods”. In: *International Journal of Innovative Research in Computer and Communication Engineering* 2.2 (2014).

[24] Junzo Watada et al. “Human Tracking: A State-of-Art Survey”. In: *Knowledge-Based and Intelligent Information and Engineering Systems: 14th International Conference, KES 2010, Cardiff, UK, September 8-10, 2010, Proceedings, Part II*. Ed. by Rossitza Setchi et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 454–463. ISBN: 978-3-642-15390-7.

[25] Alper Yilmaz, Omar Javed, and Mubarak Shah. “Object Tracking: A Survey”. In: *ACM Computing Surveys* 38.4 (2006), pp. 1–45.

[26] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

[27] L. Čehovin, A. Leonardis, and M. Kristan. “Visual Object Tracking Performance Measures Revisited”. In: *IEEE Transactions on Image Processing* 25.3 (Mar. 2016), pp. 1261–1274. ISSN: 1057-7149. DOI: `10.1109/TIP.2016.2520370`.

[28] C. Garcia and M. Delakis. “Convolutional face finder: a neural architecture for fast and robust face detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.11 (2004), pp. 1408–1423.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems*. Vol. 1. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[30] Ross B. Girshick. “Fast R-CNN”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1440–1448.

[31] Ross B. Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *CoRR* abs/1311.2524 (2013).

[32] J. R. R. Uijlings et al. “Selective Search for Object Recognition”. In: *International Journal of Computer Vision* 104.2 (2013), pp. 154–171.

[33] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *CoRR* abs/1506.01497 (2015).

[34] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015).

[35] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *CoRR* abs/1512.02325 (2015).

[36] Joseph Redmon and Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *CoRR* abs/1612.08242 (2016).

[37] T. J. Broida and R. Chellappa. “Estimation of Object Motion Parameters from Noisy Images”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.1 (1986), pp. 90–99.

[38] Genevive B. Orr and Klaus-Robert Müller. *Neural Networks: Tricks of the Trade*. Springer, 1998.

[39] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research (JMLR)* 15 (2014), pp. 1929–1958.

[40] M. Everingham et al. “The Pascal Visual Object Classes (VOC) Challenge”. In: *International Journal of Computer Vision* 88.2 (2010), pp. 303–338. URL: `http://host.robots.ox.ac.uk/pascal/VOC/`.

[41] Qing Wang et al. “An Experimental Comparison of Online Object Tracking Algorithms”. In: *Proceedings of SPIE - The International Society for Optical Engineering* (Sept. 2011).

[42] Y. Wu, J. Lim, and M. H. Yang. “Online Object Tracking: A Benchmark”. In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. June 2013, pp. 2411–2418. DOI: `10.1109/CVPR.2013.312`.

[43] Y. Wu, J. Lim, and M. H. Yang. “Object Tracking Benchmark”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.9 (Sept. 2015), pp. 1834–1848. ISSN: 0162-8828. DOI: `10.1109/TPAMI.2014.2388226`.

[44] NVIDIA. *JetPack*. 2017. URL: `https://developer.nvidia.com/embedded/jetpack` (visited on 04/12/2018).

[45] NVIDIA. *Deep Learning Frameworks*. 2017. URL: `https://developer.nvidia.com/deep-learning-frameworks` (visited on 01/24/2018).

[46] Sharan Chetlur et al. “cuDNN: Efficient Primitives for Deep Learning”. In: *CoRR* abs/1410.0759 (2014).

[47] NVIDIA. *TensorRT 3: Faster TensorFlow Inference and Volta Support*. 2017. URL: `https://devblogs.nvidia.com/tensorrt-3-faster-tensorflow-inference/` (visited on 04/12/2018).

[48] TensorFlow. *Cloud Tensor Processing Units (TPUs)*. 2018. URL: `https://cloud.google.com/tpu/docs/tpus` (visited on 06/05/2018).

[49] TensorFlow. *TensorBoard: Visualizing Learning*. 2018. URL: `https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard` (visited on 06/05/2018).

[50] TensorFlow. *Tensorflow Object Detection API*. 2018. URL: `https://github.com/tensorflow/models/tree/master/research/object_detection` (visited on 05/17/2018).

[51] OpenCV-contrib. *Object tracking API*. 2018. URL: `https://github.com/opencv/opencv_contrib/tree/master/modules/tracking` (visited on 05/17/2018).

[52] Z. Kalal, K. Mikolajczyk, and J. Matas. “Forward-Backward Error: Automatic Detection of Tracking Failures”. In: *2010 20th International Conference on Pattern Recognition*. Aug. 2010, pp. 2756–2759. DOI: `10.1109/ICPR.2010.675`.

[53] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *CoRR* abs/1312.6199 ().

[54] A. Nguyen, J. Yosinski, and J. Clune. “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 427–436. DOI: `10.1109/CVPR.2015.7298640`.

[55] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *CoRR* abs/1409.0575 (2014).

[56] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *CoRR* abs/1405.0312 (2014). URL: `http://arxiv.org/abs/1405.0312`.

[57] Mark Everingham et al. “The Pascal Visual Object Classes Challenge: A Retrospective”. In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pp. 98–136. ISSN: 1573-1405. DOI: `10.1007/s11263-014-0733-5`.

[58] Computer Vision Lab at HYU. *Visual Tracker Benchmark*. 2018. URL: `http://cvlab.hanyang.ac.kr/tracker_benchmark/datasets.html` (visited on 05/13/2018).

[59] Matej Kristan et al. *The Visual Object Tracking VOT2016 challenge results*. Springer. 2016.

[60] VOT Challenge. *VOT - Dataset*. 2018. URL: `http://www.votchallenge.net/vot2018/dataset.html` (visited on 05/08/2018).

[61] TensorFlow. *Detection Model Zoo*. 2018. URL: `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md` (visited on 06/04/2018).

[62] Mark Sandler et al. “Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation”. In: *CoRR* abs/1801.04381 (2018).