FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Visual Object Detection and Tracking by the Crazyflie Quadcopter
Student:	Artem Redchych
Supervisor:	prof. RNDr. Pavel Surynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

The task is to propose new or modify the existing algorithms for the detection and tracking of simple objects that are suitable to be executed by an autonomous UAV. It is assumed that the object to be detected and tracked will be simple such as a ball and will move smoothly. The proposed algorithms will be implemented within the Crayzflie ecosystem consisting of a small quadcopter and the localization system. The tasks for the student are as follows:

1. Study the existing algorithms for visual object detection and tracking.
2. Identify algorithms and techniques that are suitable for implementation within the Crazyflie ecosystem.
3. Implement the visual detection and object tracking algorithm on the Crazyflie quadcopter using the on-board camera and the stationary localization system.
4. Perform relevant tests in the Robotic Agents Laboratory.

[1] Karol Hausman, Jörg Müller, Abishek Hariharan, Nora Ayanian, Gaurav S. Sukhatme: Cooperative multi-robot control for target tracking with onboard sensing. Int. J. Robotics Res. 34(13): 1660-1677 (2015).
[2] James A. Preiss, Wolfgang Hönig, Gaurav S. Sukhatme, Nora Ayanian: Crazyswarm: A large nano-quadcopter swarm. ICRA 2017: 3299-3304
[3] Michael Montemerlo, Sebastian Thrun:

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 27 December 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Simultaneous localization and mapping with unknown data association using fastSLAM. ICRA 2003: 1985-1991

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 27 December 2022 in Prague.

Bachelor's thesis

VISUAL OBJECT DETECTION AND TRACKING BY THE CRAZYFLIE QUADCOPTER

Artem Redchych

Faculty of Information Technology
Katedra aplikované matematiky
Supervisor: doc. RNDr. Pavel Surynek, Ph.D.
May 7, 2023

Contents

List of Figures

List of Tables

List of code listings

I want to express my gratitude to my supervisor, prof. RNDr. Pavel Surynek, Ph.D., for providing guidance throughout every step of the thesis writing process.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 7, 2023 .....

Abstract

In this thesis's literature review section, contemporary and older methods for visual object detection, tracking, and distance estimation were investigated. The main components of the Crazyflie ecosystem were also described. In the implementation part, selected methods were implemented. For object detection, the MobileNetV2-SSD model with FPNlite was used, the Kalman Filter was employed for object tracking, and the monocular distance estimation technique was utilized for distance estimation. Object detection achieved an average accuracy of 87 % and a tracking accuracy of 74 %. This work's main result is exploring the potential use of object detection and tracking methods using the Crazyflie 2.1 drone and its Loco Positioning and AI-deck modules. In the appendices of the work, one can find key components and the final implementation of the flying streamer.

Keywords object detection, object tracking, Crazyflie 2.1, machine learning, distance estimation, AI-deck

Abstrakt

V literární rešerši této práce byly prozkoumány současné i starší metody pro vizuální detekci objektů, sledování a odhad vzdálenosti. Byly také popsány hlavní složky ekosystému Crazyflie. V části implementace byly zavedeny vybrané metody. Pro detekci objektů byl použit model MobileNetV2-SSD s FPNlite, pro sledování objektů byl použit Kalmanův filtr a pro odhad vzdálenosti pomocí jediné kamery. Detekce objektů dosáhla průměrné přesnosti 87 % a přesnosti sledování 74 %. Hlavním výsledkem této práce je průzkum možného využití metod detekce a sledování objektů pomocí dronu Crazyflie 2.1 a jeho modulů Loco Positioning a AI-deck. V přílohách práce lze nalézt klíčové komponenty a finální implementaci létajícího streameru.

Klíčová slova detekce objektů, sledování objektů, Crazyflie 2.1, strojové učení, odhad vzdálenosti, AI-deck

Acronyms

CC	Crazyflie-Client
CNN	Convolution Layer
DPM	Deformable Part-Based Model
FC	Fabric Controller
FC	Fully Connected
FPN	Feature Pyramid Network
FPS	Frames Per Second
IOU	Intersection Over Union
KF	Kalman Filter
ReLU	Rectified Linear Unit
RCNN	Region-based Convolutional Neural Networks
ROI	Region Of Interest
RPN	Region Proposal Network
SSD	Single-Shot Detector
SPP	Spatial Pyramid Pooling
VOT	Visual Object Tracking
YOLO	You Only Look Once

Introduction

Over the past decade, researchers have made significant advances in computer vision. Machine learning, which experienced a significant leap in development with the emergence of advanced convolutional neural networks after 2012, has primarily driven these accomplishments. In addition, the growth and popularization of these technologies have contributed to the emergence of tinyML (machine learning on small devices). These achievements, in turn, have facilitated the application of these technologies in autonomous drones for navigation, detection, and tracking purposes. [1]

Autonomous drones employing computer vision can perform a wide range of tasks. Various domains, such as delivery, agronomy, military, surveillance, search and rescue, and many others, utilize drones. Despite considerable achievements in the field of autonomous drones, numerous challenges and limitations, both technological and legal, persist. Technical limitations include limited flight range and duration, poor quality of acquired data, and complexities in data processing. Legal restrictions involve the use of drones in urban settings. Overcoming these challenges will enhance the accuracy, safety, and productivity of autonomous drones. In this thesis, the author will focus on addressing the issue of real-time object detection and tracking in low-quality images obtained from a CrazyFlie drone. Resolving this problem will pique the interest of other students in the faculty to work with these drones. [2]

The author chose this topic because the author aspired to be the first Faculty of Informatics and Technology (FIT) student to conduct a thesis in which the author will explore the capabilities of the CrazyFlie drone in combination with the AI-deck module. Additionally, the capabilities of the CrazyFlie drone and AI-deck modules' capabilities still need to be sufficiently explored. The author's motivation stems from a desire to work on a project that could benefit people. The author sees excellent potential in autonomous drones that help people improve their lives or provide protection.

This thesis addresses computer vision tasks on the CrazyFlie drone, using the GAP8 processor found in the AI-deck module. This thesis will investigate the problem of object detection using machine learning, object tracking, and distance estimation to the target using popular algorithms appropriate for limited computational resources. On the other hand, this study will not address the problem of object localization in 3D space.

The structure of this thesis is divided into three parts. In the first part, the author describes the issues, algorithms, and techniques involved in visual object detection, tracking, and distance estimation. In the subsequent part, the author discusses the Crazyflie platform and all its components that were utilized by the author. In the final part, the author presents the implementation of the chosen algorithms, as well as demonstrates and interprets the test results obtained in the Robotic Agents Laboratory.

Goal

The main goal of this bachelor thesis is to propose new or modify existing algorithms for the detection and tracking of simple objects that are suitable to be executed by autonomous unmanned aerial vehicles.

The theoretical part aims to identify algorithms and techniques suitable for implementation within the Crazyflie ecosystem by studying existing visual object detection and tracking algorithms.

The first goal of the realization part is to implement a new or modify existing visual detection and object tracking algorithms on the Crazyflie quadcopter using the AI-deck module with an onboard camera and the stationary localization system. The second goal is to perform relevant tests on the Crazyflie quadcopter in the Robotic Agents Laboratory.

This bachelor thesis outputs the transfer of object detection and tracking techniques on a Crazyflie quadcopter that can perform visual object detection and tracking.

Chapter 1

Techniques for Object Detection and Tracking

1.1 Investigating the Problem

1.1.1 Problem analysis

Object detection is a fundamental and complex problem that allows capturing objects from real-world scenes using camera data, such as human faces, automobiles, animals, and others. The object detection task is divided into two subtasks. The first is the location of a given object in the image. The second is the classification of the object [3]. Object recognition is used in the automotive industry, video surveillance, robotics, and other fields [1, 4]. Object recognition is sometimes only the end product. Typically, detected objects are used as input data for various other methods, such as image segmentation, tracking, and image captioning [1].

1.1.2 Techniques investigation

1.1.2.1 Visual object detection

Over the past twenty years, object recognition technology has experienced rapid development. This development can be attributed to significant advances in machine learning. The work "Object Detection in 20 Years: A Survey" states that the development of object recognition can be divided into two periods: before and after the use of deep machine learning. The methods of the first period are called "*traditional object recognition methods*", while those of the second period are called "*deep learning-based object detection*". [1]

Traditional object recognition methods include:

1. Viola-Jones Detectors
2. HOG Detector
3. Deformable Part-Based Model [1]

Viola-Jones detectors were invented in 2001 by P. Viola and M. Jones. With this method, real-time human face recognition was achieved. As the authors note in their article "Rapid object detection using a boosted cascade of simple features", the main contribution of their discovery consists of three aspects:

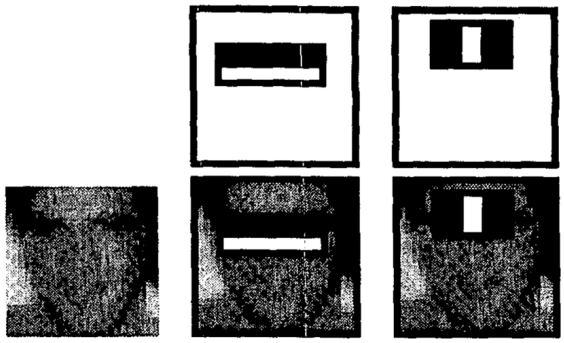
1. Integral image – a new image representation that allows for fast feature evaluation.

2. A method for building a classifier by selecting a small number of essential features using AdaBoost.

3. A method for cascading classifier construction, which significantly accelerates the detection speed of an object in an image. [5]

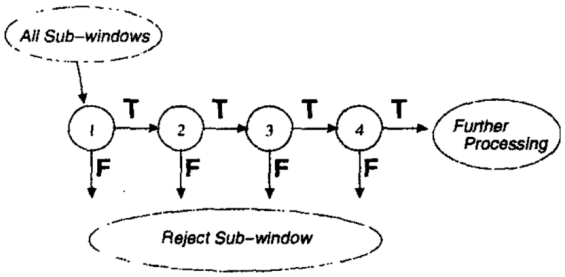
The working principle of the algorithm is as follows. First, an integral image is generated on the basis of the pixel values of the image. The calculation is done only once, greatly accelerating the calculations. Then a sliding sub-window of 24 by 24 pixels is created (the size may vary), in which the required features are checked in a cascading manner; the example is shown in 1.1. [5]

Figure 1.1 Haar features, selected for face detection [6]



If the sub-window passes the first check, it is tested on the next classifier, and so on. If the sub-window does not pass the test on any classifier, that sub-window is discarded, and another sub-window is checked. This cascading classification method enables faster computation time, as most sub-windows will be discarded in the first and simpler classifiers, with more complex ones being in the last steps. If a sub-window passes through all classifiers, a positive result for face detection is produced, indicating that a face is present. The principle of cascading classifier operation is schematically shown in figure 1.2. [5]

Figure 1.2 Visualization of cascade detection [6]



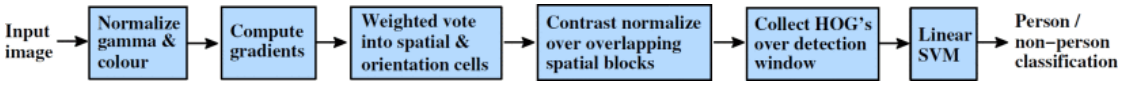
At first glance, the working principle of the Viola-Jones detector is relatively simple. However, computing power that was considered immense at that time was required for the calculations. [5]

The HOG Detector, which has been an essential component of numerous object detection algorithms over the years, was introduced by Dalal and Triggs in 2005 to enhance pedestrian detection methods. Although the main objective of its development was the detection of people, it can also be used to detect other objects. [1]

The working principle is schematically illustrated in Figure 1.3. "*The method is based on evaluating well-normalized local histograms of image gradient orientations in a dense grid. The fundamental idea is that the appearance and shape of a local object can often be characterized relatively well by the distribution of local intensity gradients or edge directions, even without precise knowledge of the corresponding gradient or edge positions*" [7]. In practice, the detection process unfolds as follows:

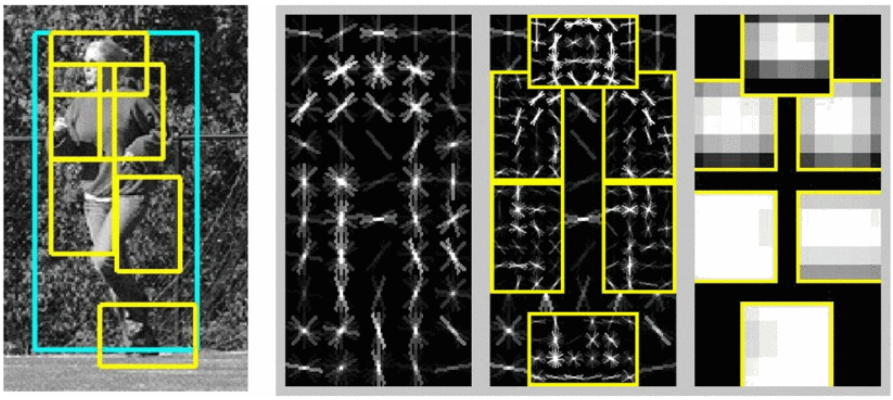
1. A detection window, typically 64x128 pixels in size (which can be adjusted for smaller images), is selected.
2. This window traverses all possible parts and scales of the image during the detection process.
3. The gradient is calculated for each pixel within the given window.
4. The image is divided into 8x8-pixel blocks and a 9-bin histogram is computed for each of them. Subsequently, cells are grouped into 2x2, 4x4, and 8x8 blocks.
5. On the basis of the blocks obtained, a feature vector is computed, which is then normalized and combined with other vectors to create the HOG feature.
6. Finally, the HOG features are fed into a linear SVM classifier, which responds to whether a human is present in the image [7].

Figure 1.3 An overview of our feature extraction and object detection chain [8]



The Deformable Part-Based Model (DPM) was proposed by Felzenszwalb in 2008 and embodies traditional object detection methods and extends the HOG detector. A vital feature of this method is its "divide and conquer" approach, in which object detection is divided into several smaller tasks. For example, one could first detect individual body parts such as the head, arms, and legs to detect a person. Upon detecting all these parts, it is then possible to accurately determine whether a person is present in the image, as illustrated in Figure 1.4. [1, 9]

Figure 1.4 Example detection obtained with the person model [10]



The detection system employs a sliding-window principle. The object model consists of a primary filter and part models. Each part model includes a spatial model, which indicates where a specific part of the object may be located, and a corresponding filter. Simplified object detection can be divided into the following steps:

1. A feature map is constructed using the HOG method.
2. The sliding window traverses each region of the feature map, and the filters are used to check for template matches. The primary template represents the entire object, while the others represent object parts.
3. For each region, a detection score is calculated based on the primary and auxiliary filters.
4. The detection result is determined based on the detection score. [9]

Although most modern object detection methods surpass DPM in terms of accuracy, many incorporate the principles presented within them. In 2010, the authors of the method received a ”lifetime achievement” award from PASCAL VOC for their significant contributions to the field. [1]

1.1.2.2 Machine learning for computer vision

Traditional object detection methods are based on hand-crafted features and shallow learning architectures, which in turn impose limitations on their capabilities. Deep learning methods significantly outperform them by being able to learn semantics and detect deeper features. [11, 1]

A convolutional neural network (CNN) is a kind of artificial neural network that employs numerous perceptrons to examine image inputs. It utilizes learnable weights and biases for different parts of images, enabling them to be distinguished from one another. One benefit of using a CNN is that it takes advantage of local spatial coherence in input images, allowing for shared parameters, and thus fewer weights. This approach is notably efficient in terms of memory and complexity. The fundamental components of a convolutional neural network include the following:

1. Convolution layer – In the convolutional layer, a kernel matrix is moved across the input matrix to generate a feature map for the subsequent layer. This is done by performing a mathematical operation called convolution, which involves sliding the kernel matrix over the input matrix and executing element-wise matrix multiplication at each position, with the results summed onto the feature map.
2. Nonlinear activation functions (ReLU) – The activation function is a nonlinear transformation applied to the input signal, following the convolutional layer. For example, the rectified linear unit (ReLU) activation function outputs the input if it is positive and zero if it is negative.
3. Pooling Layer – A problem with feature maps from convolutional layers is that they track exact feature positions, so small changes in the input image can create very different maps. To solve this problem, a pooling layer is used after the nonlinearity layer, which helps to keep the output mostly the same even if the input shifts slightly.
4. Fully Connected Layer – In a convolutional neural network, the final pooling layer's output serves as input for the fully connected layer(s). Fully connected implies that each node in one layer is connected to every node in the subsequent layer. [12]

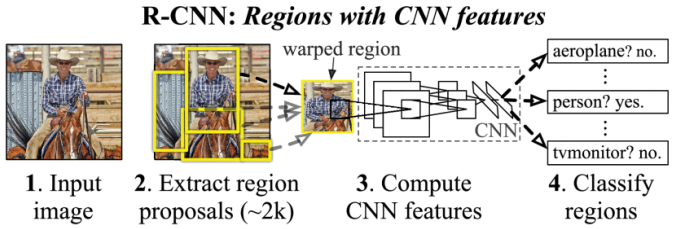
The object detection task consists of localizing the object and then classifying it. Therefore, two groups of detectors are distinguished: ”two-stage detectors” and ”one-stage detectors.” The former separately addresses the localization and classification tasks, while the latter solves these tasks simultaneously. The first group includes the following: Traditional object recognition methods include:

1. RCNN
2. SPPNet

3. Fast RCNN
4. Faster RCNN
5. Feature Pyramid Networks (FPNs) [11, 1]

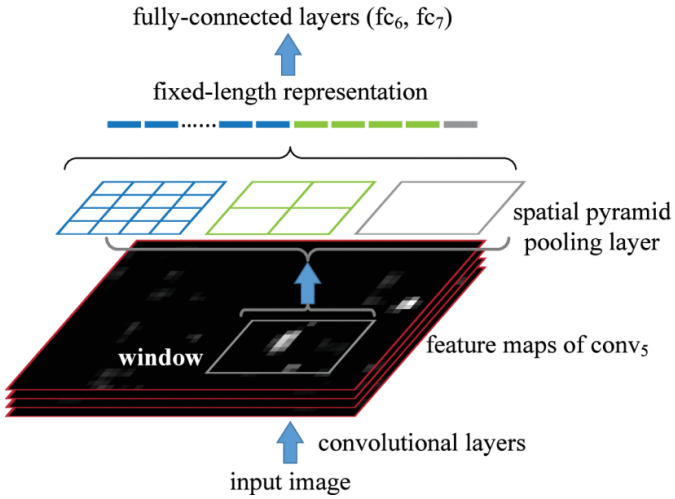
The RCNN is based on a simple idea that is shown in 1.5 initially employs a selective search to identify regions of the image, which are then passed to CNN for feature extraction. These features are then sent to a linear SVM for object classification within the region. However, despite the significant improvements in object detection achieved by RCNN, there is a substantial drawback in redundant feature computations in overlapping regions, resulting in inefficient calculations. [11, 1]

Figure 1.5 Architecture of R-CNN [13]



SPPNet resolved the issue of the previous method, which limited the input image sizes and required feature extraction to be performed in each detection iteration. SPPNet proposed a solution to these problems by extracting features only once for the entire image and adding a Spatial Pyramid Pooling (SPP) layer as shown in 1.6, which allows feature maps to be generated independently of the image size. With almost 20 times the acceleration of RCNN without sacrificing detection accuracy, SPPNet significantly improved computational efficiency. However, the problems of multistep training and the fact that SPPNet only fine-tunes its fully connected layers, ignoring all previous layers, remained. [11, 1]

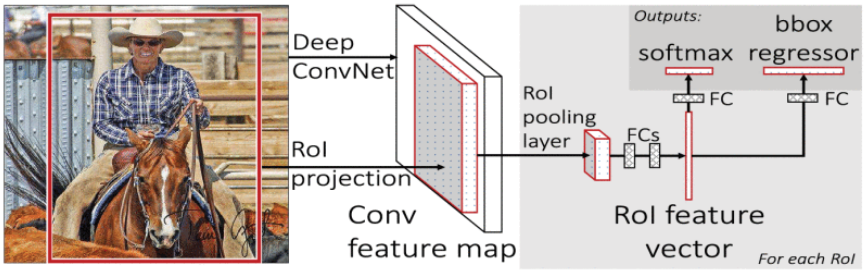
Figure 1.6 SPPnet architecture [13]



Fast RCNN was introduced in 2015 by Girshick et al. and presented an improvement over RCNN and SPPNet. Schematically depicted in 1.7, the working principle involves performing feature extraction on the entire image, unlike its predecessors. Subsequently, a fixed-length

feature vector is created from these features using Region of Interest (ROI) pooling, which is a specific form of Spatial Pyramid Pooling. Each vector is then passed to the Fully Connected (FC) layers, sending their output to a classifier and bounding box regressor. Despite addressing the issues of its predecessors, computational speed problems persisted due to the inefficiency of proposal detection. [11, 1]

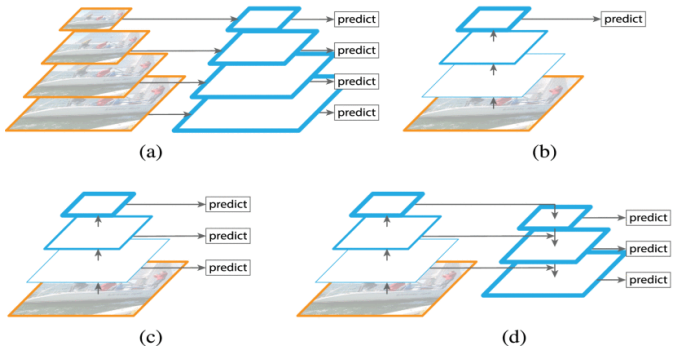
Figure 1.7 Architecture of Fast R-CNN [13]



Faster RCNN was introduced shortly after Fast RCNN by Ren et al. This method was the first capable of detecting objects almost in real time. This inference speed was made possible through a Region Proposal Network (RPN), which efficiently locates object-containing regions of the image (regardless of object class) and then forwards them along with the feature map to the box-classification and box-regression layers. Limitations of this approach include the resource-intensive nature of the RPN, which can only identify object-shaped regions and struggle with large objects of size and shape. [11, 1]

Feature Pyramid Networks (FPNs) were introduced by Lin et al. in 2017. Before this, the Feature Pyramid principle had been utilized to enhance the detection of images of varying sizes, but its implementation by the FPN predecessors could have been more efficient. It is also worth noting that most methods construct detection from the bottom up, which means that increasingly complex features are composed of more minor features that influence detection. FPN was the first to adopt an alternative principle, as shown in 1.8, whereby features at lower levels could significantly affect object detection. This novel approach to feature processing constitutes the primary contribution of FPN. FPN has become a fundamental component of many contemporary architectures. [11, 1]

Figure 1.8 Main concern of FPN [13]



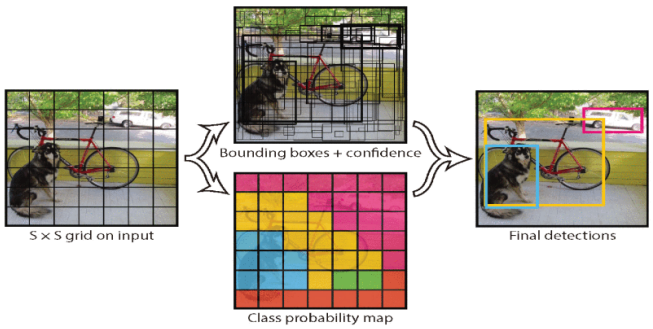
Considering their complexity and slow speed, two-stage detectors are rarely used to solve contemporary problems. Although one-stage detectors may have lower accuracy, they are frequently used due to their speed and ability to function on mobile devices [11, 1].

The ”One-stage detectors” group includes:

1. You Only Look Once (YOLO)
2. Single-Shot Multibox Detector (SSD)
3. RetinaNet [11, 1]

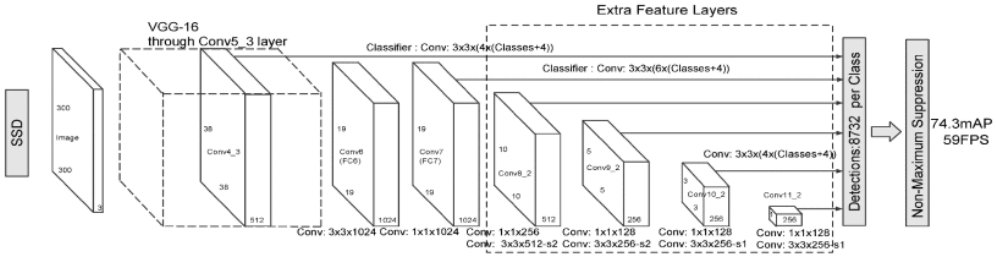
You Only Look Once was introduced by Joseph et al. in 2015. The main idea, illustrated in 1.9, involves dividing the image into a grid of S x S where each cell is responsible for predicting whether an object is present and the confidence of the algorithm in this prediction. As a result, bounding boxes are created. Then, the likelihood of each class being present in a cell is calculated independently of the number of bounding boxes, and only cells containing a class influencing the creation of the bounding box. [11, 1]

Figure 1.9 Main concern of YOLO [13]



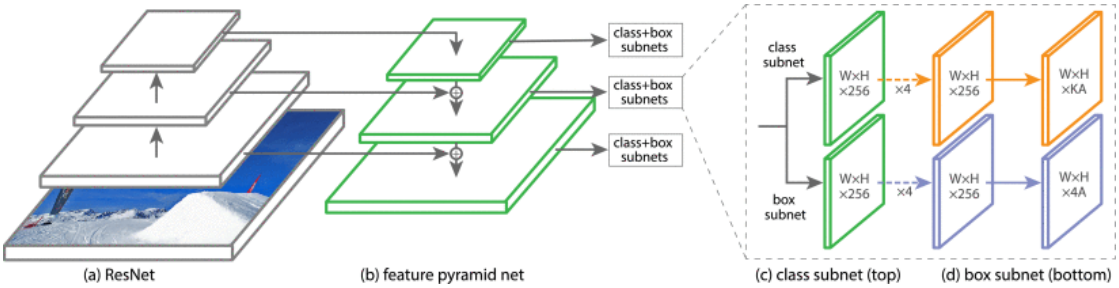
Single-Shot Multibox Detector was introduced by Liu et al. in 2015. The main contribution of this method was the significantly improved accuracy of the one-stage detectors. SSD aims to improve the detection of small objects in groups, an issue with which YOLO needed help. The SSD architecture, depicted in 1.10, consists of a base CNN with additional convolutional layers added at the end, which helps to identify candidates for the bounding box with greater precision. Eight thousand seven hundred thirty-two bounding box candidates are obtained from the convolutional layers' output. Subsequently, Nonmaximum Suppression filters only 200 bounding boxes for each class. [11, 1]

Figure 1.10 The SSD architecture [13]



RetinaNet was introduced by Lin et al. in 2017. All previous one-stage detector methods shared a common drawback: the calculation process generated excessive (often useless) bounding box candidates, negatively impacting training results. RetinaNet not only achieved but also surpassed the accuracy of two-stage detectors by addressing this issue. This detection accuracy was achieved by applying a new error calculation method during training, which helps focus on negative results. The RetinaNet schema is shown in 1.11. [14, 11, 1]

Figure 1.11 The RetinaNet schema [13]



1.1.2.3 Visual object tracking

Visual object tracking (VOT) is an essential part of computer vision. With powerful computers, good quality and affordable video cameras, and the need for automatic video analysis, people are more interested in object tracking algorithms. There are three main steps in analyzing videos: finding interesting moving objects, following these objects from one frame to another, and studying their movement to understand their behavior. Therefore, object tracking is beneficial for tasks like:

1. Traffic monitoring
2. Industrial robotics
3. Vehicle tracking
4. Vehicle navigation [15, 16]

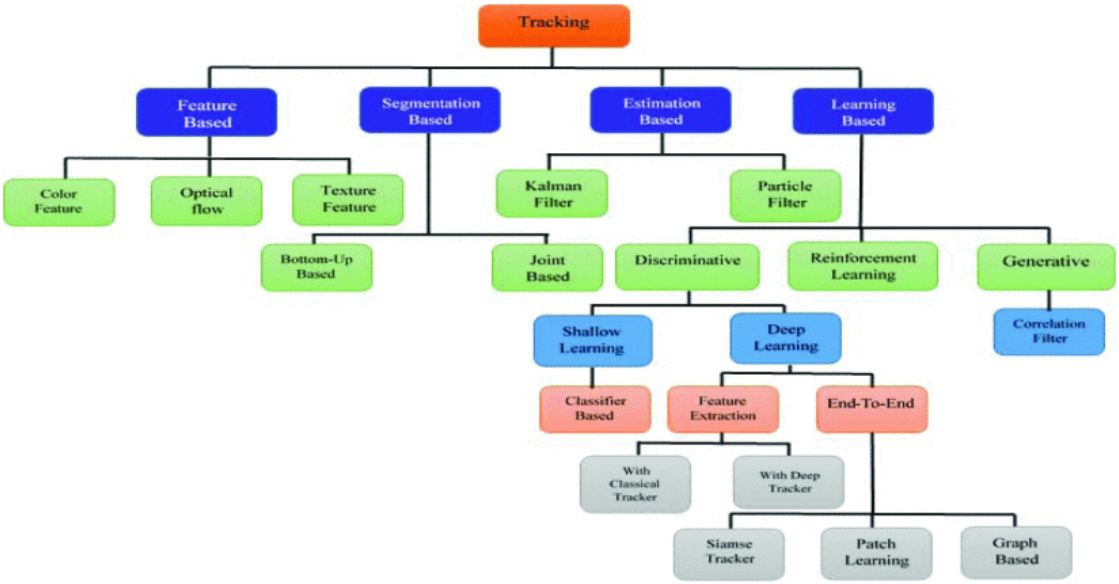
In its most basic form, tracking refers to the process of determining an object's path in the image plane as it moves throughout a scene. In essence, a tracker assigns consistent labels to objects being tracked across various video frames. Furthermore, depending on the tracking domain, a tracker can also provide object-specific information, such as orientation, area, or shape [15]. Researchers have put a lot of work into visual object tracking (VOT) for the past 40 years. However, it remains an open area for computer vision research because of various challenges. These issues can be described as follows:

1. Occlusion – This problem occurs when a target is partially or fully hidden by another object, presenting a common challenge during tracking. There is no universal technique to address it, so strategies are chosen on the basis of the target's nature and the tracking environment.
2. Changing appearance – Targets can change appearance during motion, so the model must adapt for long-term tracking. The stability vs. plasticity dilemma is to find the right balance between stability and adaptability.
3. Changing target size in image – As a target moves closer or farther from the camera, its size in the image changes, so tracking strategies need to account for this.
4. Noise in the image – If the image from the target scene has noise, some preprocessing is needed to remove it. [15, 16]

Object tracking methods can be classified in various ways. The authors of "Object Tracking Methods:A Review" described different categories of object detection techniques and used classification, which is shown in figure 1.12

Feature-based tracking is a simple method where unique features like color, texture, and optical flow are extracted to distinguish objects. After extraction, the most similar object in the

Figure 1.12 Classification of object tracking methods [17]



next frame is identified using a similarity criterion. The challenge lies in extracting unique and reliable features to differentiate the target from other objects. The feature-based methods use the following features:

1. Color – Color features represent an object's appearance and can be used in various ways, such as color histograms, which display the distribution of colors and the pixel count for each color in an image. However, color histograms only consider color, not shape or texture, so two different objects may have the same histogram.

2. Optical flow – Optical flow refers to the perceived motion of brightness patterns in an image, which may result from lighting changes without actual movement. Optical flow algorithms measure the displacement of these patterns between frames. Dense optical streaming algorithms calculate displacement for all pixels, while light-flow algorithms estimate displacement for a select number of pixels in an image.

3. Texture – Texture features, derived from image preprocessing techniques, represent repeated patterns or structures. They can be combined with color features to better describe an image or region. Gabor wavelets, which are invariant to illumination, rotation, scale, and translation, are a popular texture feature. [18, 19]

Segmenting foreground objects from a video frame is essential and critical for visual tracking. The foreground segmentation separates moving objects from the background scene. To track these objects, they must be distinguished from the background. There are the following object tracking methods based on segmentation:

1. Bottom-Up based method – In this tracking approach, two distinct tasks are performed: foreground segmentation followed by object tracking. Foreground segmentation involves low-level segmentation to extract regions in all frames. Then, features are extracted from foreground regions and used for tracking based on those features.

2. Joint Based Method – In the bottom-up method, foreground segmentation and tracking are separate tasks, leading to segmentation errors that propagate and cause tracking errors.

To address this problem, researchers combined the foreground segmentation and tracking method, which improved the tracking performance. [18, 19]

Estimation methods turn tracking into an estimation problem by representing an object with a state vector that describes its dynamics, such as position and velocity. Using Bayesian filters, these methods continuously update the target's position based on sensor data through a two-step process: prediction and updating. The prediction estimates the target's next position, while updating refines that position with current observations, and this cycle is repeated for each video frame. There are following object tracking methods based on estimation:

1. Kalman filter – The Kalman filter is utilized in object tracking by designing a dynamic model for target movement. It estimates the position of a linear system with Gaussian errors. If the dynamic models are nonlinear, the Kalman filter is not suitable, and alternatives such as the extended Kalman filter are employed instead.

2. Particle filter – Particle filters help track objects in complex situations using particles and probabilities. They can handle challenges like non-Gaussian noise but sometimes need re-sampling to fix issues with high probability particles. [18, 19]

Learning-based methods for tracking learn features and appearances of targets to predict their positions in future frames. These methods can be categorized into generative, discriminative and reinforcement learning approaches. [18, 19]

Generative methods concentrate on searching areas similar to the object, with Correlation Filter-based trackers being a prime example. The main concept involves estimating an optimal image filter to generate an ideal output in the input image. The target is identified in the first frame, and the filter is trained on it. Then, at each step, the patch is cropped to its predicted position for tracking, features are extracted, and a spatial confidence map is obtained. Finally, the new target position is predicted and the correlation filter is updated accordingly. [18, 19]

Discriminative trackers often treat tracking as a classification issue, differentiating the target from the background (Siamese tracking, Patch learning tracker,Graph-based tracker). [18, 19]

Reinforcement learning involves an agent that interacts with the environment through trial and error to select the optimal action to achieve a goal. Some studies use reinforcement learning for tracking, such as an approach divided into a matching network and a policy network. Given a frame, a search image is cropped based on previous target information. Using appearance templates from previously tracked frames, the matching network generates prediction maps. The policy network then scores each map, selecting the one with the highest score for target tracking. The policy network is trained to handle various situations. [18, 19]

1.1.2.4 Distance Estimation Techniques

Determining the position of an object using a camera is an essential and complex task in computer vision. This work will investigate the following techniques that can be employed to calculate the distance between the camera and the object:

1. Using a single camera

2. Using two cameras (stereo vision)

3. Using machine learning

In work "A Monocular Vision Advance Warning System for the Automotive Aftermarket," a simple method for determining distance is described. For the calculations, it is necessary to have information about the camera focal length, the height at which the camera is positioned, the size of the object in the image, and its actual size. This is achieved using geometry and, in particular, the method of similar triangles. With this formula, the distance can be calculated quite accurately. [20]

In the article "Depth Estimation Using Monocular Camera," the authors discuss how the distance determination from the camera to the object using the previous method encounters the issue of perspective distortion, which may lead to inaccurate results. In their work, they describe methods for eliminating horizontal and vertical errors. Vertical errors arise along the camera's optical axis, increasing when the object is closer to the camera. Horizontal errors, on the other hand, are orthogonal to the camera's optical axis and increase as the object moves away from the camera. This method helps to achieve greater accuracy in determining the distance, although not very significantly. [21]

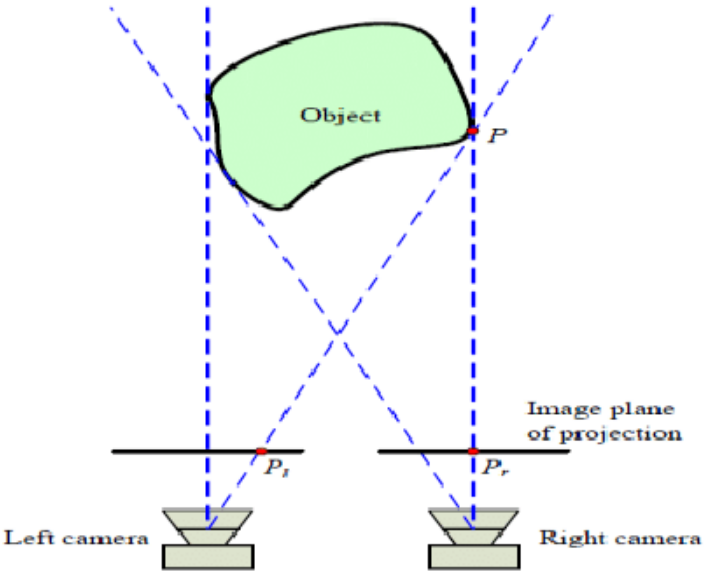
The stereo vision method is a technique that uses two or more cameras to recreate a 3D scene. This process is usually divided into two steps:

1. Corresponding problem – Calculate how the pixels are shifted to the respective points in the other image for each point in the image. The results are stored in the disparity map.

2. Triangulation – Calculate 3D coordinates using the disparity map, camera positions, focal length, and orientation. [22]

In the work "Stereo Vision Distance Estimation Employing SAD with Canny Edge Detector", the authors describe the stereo vision principle using two cameras. A schematic representation of the cameras and the object is shown in figure 1.13.

Figure 1.13 Cameras position in stereo vision [23]



Using the principle of triangulation, it is possible to calculate the distance to the object from a stereo system. The fundamental concept is that the disparity between the pixel values at a given point in the images will be greater if an object is closer. On the contrary, the greater the distance, the smaller the disparity. It is important to note that the cameras must be positioned at the same level. [22]

The task of determining the distance from an object to the camera can also be solved using machine learning. In work "Multi-DisNet: Machine Learning-Based Object Distance Estimation from Multiple Cameras," the authors trained a model to determine distance based on images of objects with bounding boxes. The closer the object, the larger the rectangle. Using this approach, we can achieve good accuracy. [24]

1.2 Justification and Description of Selected Algorithms

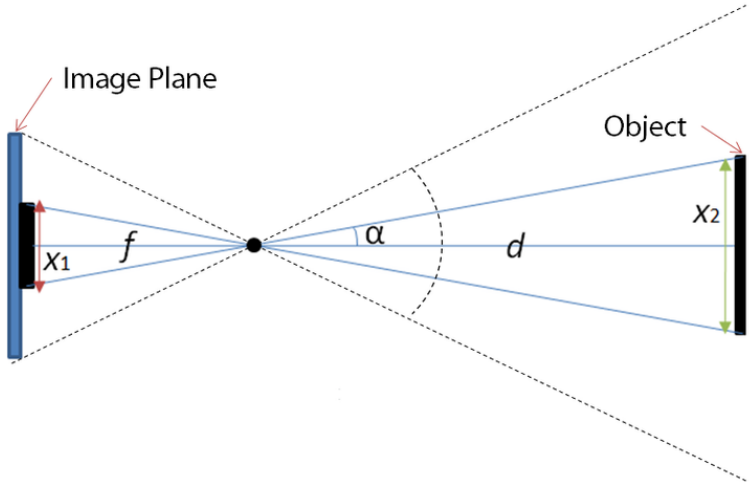
1.2.1 Monocular distance estimation technique

Each of the methods described in 1.1.2.4 has its advantages and disadvantages. The benefits of stereo vision and monocular methods are their speed. Calculations are performed in real time with minimal delay. The main drawback of the stereo system is that the cameras must be parallel in the same plane; deviations from the correct configuration may cause errors in the calculations.

Since a small drone with a single camera is used in this work, the stereo vision method could be utilized. The first picture would need to be taken at one place, and then the drone would be moved to another position, where the second image would be taken. Unfortunately, this is not feasible because the location and movements of the drone need to be more precise, which could lead to significant errors in the system's operation.

Considering the advantages and disadvantages of various distance determination methods, the author chose to use a monocular system because of its simplicity, speed, and suitability under laboratory conditions. The author will now describe the working principle of this system.

Figure 1.14 Schematic diagram of the imaging geometry [25]



The height of a projected object located in front of a camera at a distance d will be denoted as $X1$, where $X1$ is given by the formula:

$$X1 = \frac{f \cdot X2}{d} \tag{1.1}$$

Here, X2 represents the actual height of the object and f denotes the focal length of the camera. This formula was derived using the similarity of triangles: $\frac{X1}{f} = \frac{X2}{d}$. This concept is schematically illustrated in 1.14. Knowing the values of $X1$, $X2$ and f, the distance d can be determined using the previous formula. [26]

$$d = \frac{f \cdot X2}{X1} \tag{1.2}$$

1.2.2 Kalman filter in visual object tracking

Considering the objectives and conditions of the experiments for tracking an object, specifically that the object will move at a low speed and will not exhibit sudden changes in its trajectory or velocity, the author decided to employ the classical Kalman Filter to address the posed challenges. This method has demonstrated its speed and effectiveness in single-object tracking. Subsequently, the author will describe the principles and technical details of the Kalman Filter, drawing on the works "Visual object tracking – classical and contemporary approaches" and "Object tracking: A survey." [15, 16]

The Kalman filter(KF) is a statistical parametric recursive algorithm designed for discrete time systems, specifically linear dynamic systems. It uses a state space representation and operates in two steps: prediction and correction. The prediction step employs the state model to estimate the new state of the variables, while the correction step refines the estimates. The filter assumes the state to be distributed by a Gaussian. State space representation is shown in the following equations [15, 16]:

$$\boldsymbol{X}_{n+1} = \boldsymbol{\Phi}\boldsymbol{X}_n + \boldsymbol{U}_n, \tag{1.3}$$

$$\boldsymbol{Y}_n = \boldsymbol{M}\boldsymbol{X}_n + \boldsymbol{V}_n, \tag{1.4}$$

The state vector is represented by $\boldsymbol{X}_n$, with $\boldsymbol{\Phi}$ as the state transition matrix, $\boldsymbol{U}_n$ as the system noise vector, $\boldsymbol{V}_n$ as the observation noise vector, $\boldsymbol{Y}_n$ as the measurement vector and $\boldsymbol{M}$ as the observation matrix. [16]

The Kalman filter estimates the dynamic system states, accounting for noisy measurements (Gaussian noise) and model uncertainty. It operates in a prediction-correction cycle, using observed states to correct predicted states and update its gain matrix for improved future predictions, as outlined in equations (1.5)-(1.10). [16]

$$\boldsymbol{X}^*_{n|n} = \boldsymbol{X}^*_{n|n-1} + \boldsymbol{K}_n(\boldsymbol{Y}_n - \boldsymbol{M}\boldsymbol{X}^*_{n|n-1}) \tag{1.5}$$

In this context, $\boldsymbol{X}^*_{n|n}$ denotes the posterior measurement, $\boldsymbol{X}^*_{n|n-1}$ signifies the prior measurement, and $\boldsymbol{K}_n$ represents the Kalman gain matrix, which is defined as [16]:

$$\boldsymbol{K}_n = \boldsymbol{S}^*_{n|n-1}\boldsymbol{M}^T[\boldsymbol{R}_n + \boldsymbol{M}\boldsymbol{S}^*_{n|n-1}\boldsymbol{M}^T]^{-1} \tag{1.6}$$

In this case, $\boldsymbol{R}_n$ refers to the observation noise covariance calculated using equation (1.7), while $\boldsymbol{S}^*_{n|n-1}$ denotes the predictor error covariance defined by equation (1.8) where E is the expected value. [16]

$$\boldsymbol{R}_n = \mathrm{COV}(\boldsymbol{V}_n) = E[\boldsymbol{V}_n\boldsymbol{V}_n^T] \tag{1.7}$$

$$\boldsymbol{S}^*_n|_{n-1} = \mathrm{COV}(\boldsymbol{X}^*_n|_{n-1}) = \boldsymbol{\Phi}\boldsymbol{S}^*_{n-1|n-1}\boldsymbol{\Phi}^T + \boldsymbol{Q}_n \tag{1.8}$$

$$\boldsymbol{S}^*_{n-1|n-1} = \mathrm{COV}(\boldsymbol{X}^*_{n-1|n-1}) = [\boldsymbol{I} - \boldsymbol{K}_{n-1}\boldsymbol{M}]\boldsymbol{S}^*_{n-1|n-2} \tag{1.9}$$

In this context, $\boldsymbol{Q}_n$ represents the noise covariance matrix, which is calculated using equation (1.10) [16].

$$\boldsymbol{Q}_n = \mathrm{COV}(\boldsymbol{U}_n) = E[\boldsymbol{U}_n\boldsymbol{U}_n^T] \tag{1.10}$$

During the tracking process, the Kalman Filter operates in two modes:

1. Normal tracking mode
2. Occlusion mode [16]

During normal tracking mode, KF predicts the next target coordinates in the image plane based on measurements to define the optimal search window. During occlusion mode, KF disregards the measured value and uses its predicted value for the next state prediction, effectively handling short-term occlusion. [16]

Figure 1.15 (a) Regular tracking, and (b) the challenging problem of occlusion, with the Kalman Filter effectively managing the situation [27]

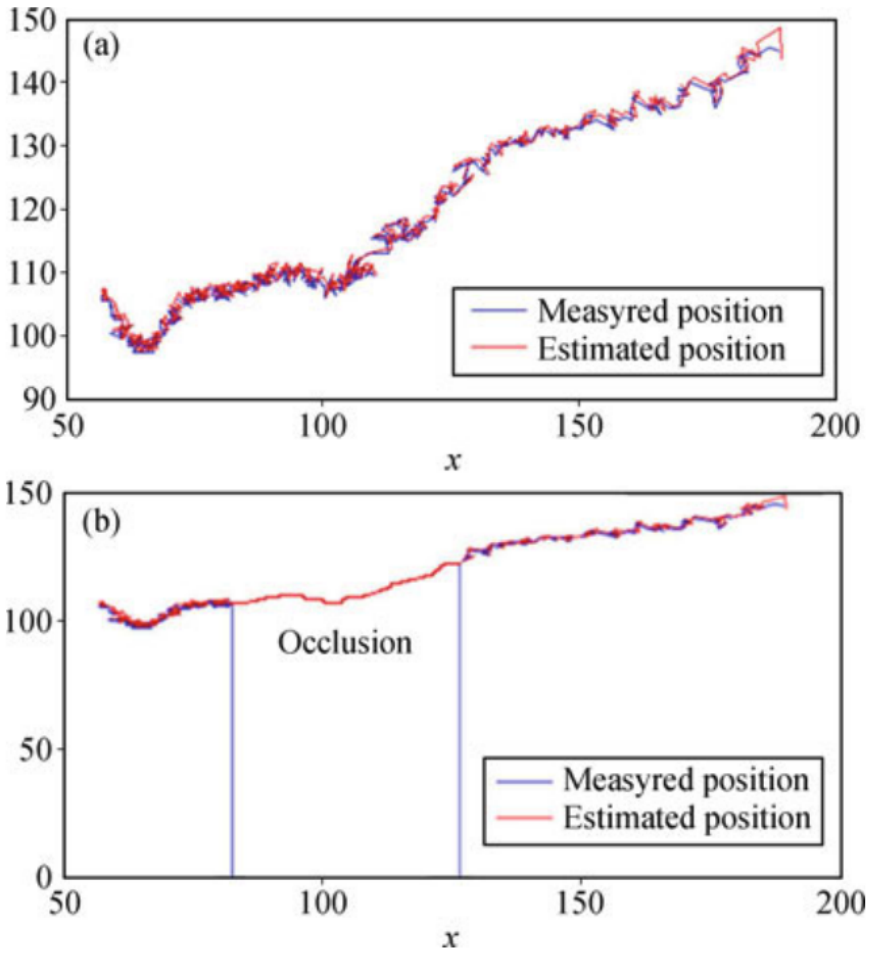


Figure 1.15(a) demonstrates the normal tracking mode, while Figure 1.15(b) displays the occlusion mode during tracking, showing that KF effectively addresses the occlusion issue. [16]

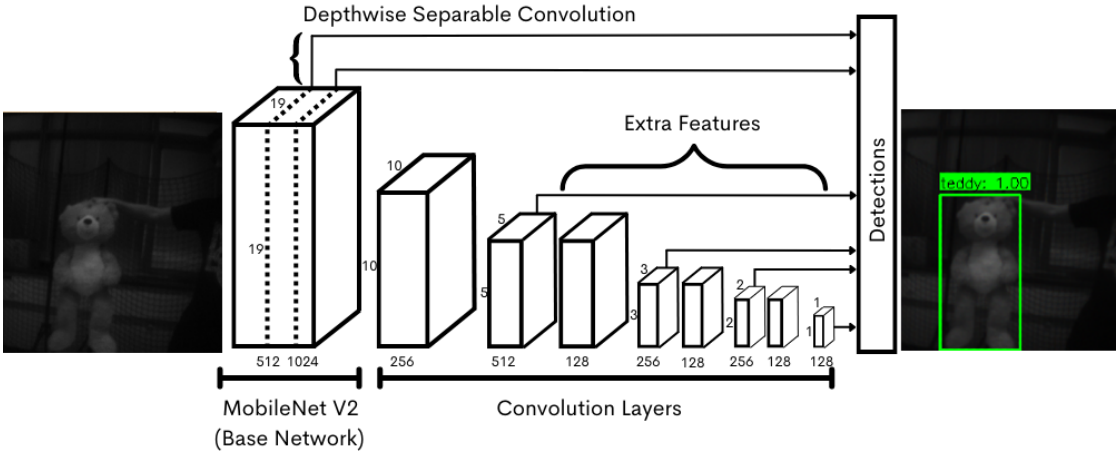
1.2.3 Visual object detection using machine learning

The author considered three factors to select the optimal object detection method for this work. The first factor is the detection speed. It is necessary to detect objects in real-time, specifically at the frame rate captured by the camera, which is 60 frames per second(FPS). The second factor is the accuracy of the detection. As the ultimate goal is to track the object after its detection, sufficient accuracy is required. The third factor is the mobility of the architecture, meaning that object detection should occur on a device with limited computational capabilities. Finally, a secondary factor is the ease of implementation and training of the detection model.

Based on all these criteria, the MobileNetV2-SSD model with FPNilte was chosen. Section

1.1.2.2 described the SSD model that uses VGG-16 as the backbone model. Since this model requires significant storage space and has a relatively slower computation speed, the author decided to use the lighter and faster MobileNetV2 model. To achieve more accurate object detection, FPN was also employed. The schematic of the final architecture is shown in figure 1.16. [28]

Figure 1.16 MobileNetV2-SSD architecture [29]



For a complete understanding of the chosen architecture, it is essential to comprehend how MobileNetV2 works. In the description of this network, the author relies on the work "MobileNetV2: Inverted Residuals and Linear Bottlenecks", which first introduced this network architecture. MobileNetV2 achieves high accuracy and computational speed due to three key factors:

1. Depthwise Separable Convolutions
2. Linear Bottlenecks
3. Inverted Residuals [30]

Depthwise Separable Convolutions are employed in various architectures because of their efficiency, which is why MobileNetV2's authors decided to utilize them when creating the new architecture. Its principle involves separating the full convolutional operator into two distinct layers. In the first, a convolutional filter is applied to each input channel, called depthwise convolution. In the second layer, a 1×1 convolution calculates features "*through computing linear combinations of the input channels*". Generally, to calculate one output tensor L_i, a convolutional filter $K \in \mathbb{R}^{k \times k \times d_i \times d_j}$ is applied to an input tensor with dimensions $h_i \times w_i \times d_j$. The number of operations is illustrated in:

$$h_i \cdot w_i \cdot d_i \cdot d_i \cdot k \cdot k \tag{1.11}$$

Using Depthwise Separable Convolutions can significantly reduce the number of computational operations, specifically:

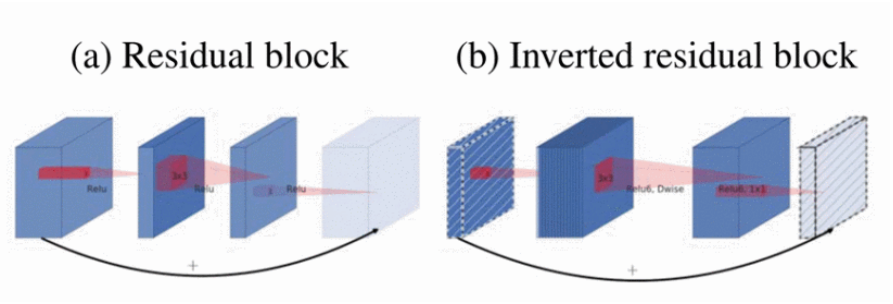
$$h_i \cdot w_i \cdot d_i(k^2 + d_j) \tag{1.12}$$

"*Effectively depthwise separable convolution reduces computation compared to traditional layers by almost a factor of* k^2." By using a filter size of 3, the number of computational operations can be reduced by 8 to 9 times at the expense of a slight decrease in accuracy. [30]

In their work, the authors discuss the properties of activation tensors in deep neural networks and how they form a "manifold of interest" that can be embedded in low-dimensional subspaces. They highlight the benefits of using linear bottleneck layers in convolutional blocks to optimize neural architectures, as reducing dimensionality can improve efficiency and accuracy. However, the authors also note that non-linear transformations like ReLU can result in information loss and limit the network's classification capabilities. They highlight that ReLU can maintain complete information about the input manifold only if it lies in a low-dimensional subspace of the input space. Based on these insights, the authors suggest using linear bottleneck layers in convolutional blocks to maintain information while introducing complexity into the set of expressible functions. [30]

Bottleneck blocks are similar to residual blocks, which have an input, several bottlenecks, and then an expansion. However, based on the idea that bottlenecks have all the needed information and the expansion layer is just a supporting detail for changing the tensor, the authors connected the bottlenecks directly with shortcuts. Figure 1.17 shows the differences between the designs. The reason for adding shortcuts is similar to why we use regular residual connections: to help gradients flow through many layers. The inverted design saves more memory and works a bit better in our experiments. [30]

Figure 1.17 The difference between residual block and inverted residual [31]



After describing the theory and principles of operation of this model, the author outlines the architecture of the MobileNetV2 model. The authors of MobileNetV2 say "*the basic building block is a bottleneck depth-separable convolution with residuals.*" 1.18 displays the specific structure of this block. The MobileNetV2 architecture consists of an initial full convolution layer with 32 filters, followed by 19 residual bottleneck layers outlined in 1.19. ReLU6 is employed as the non-linearity due to its stability in low-precision calculations. The network authors consistently use a 3×3 kernel size, typical in contemporary networks, and apply dropout and batch normalization during training. [30]

Figure 1.18 Bottleneck residual block transforming from k to k' channels, with stride s, and expansion factor t [31]

Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

The authors of MobileNetV2 managed to create a straightforward network architecture that can then be used to create other models. This was achieved through the building unit, which, thanks to its properties, can work efficiently on devices with limited computational capabilities. [30]

Figure 1.19 The architecture of MobileNetV2 [31]

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

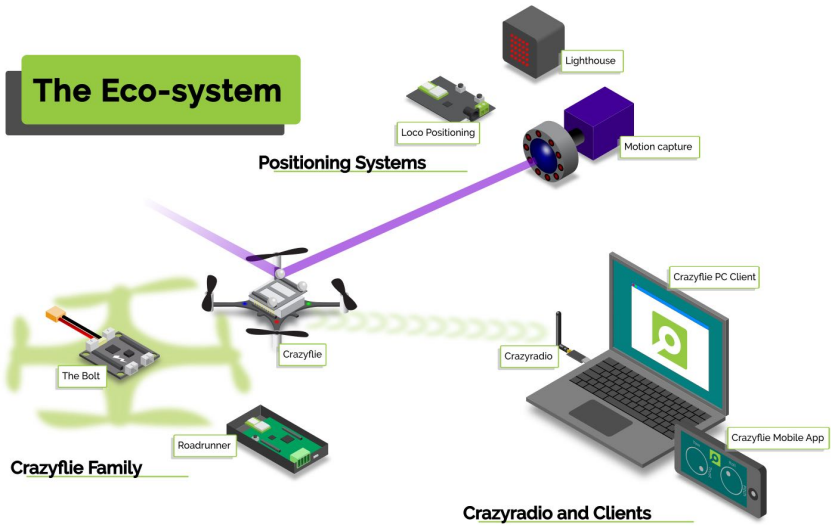
Chapter 2
Creazyflie platform

2.1 Description

Crazyflie is a versatile open source platform created by Bitcraze in 2011. It originated from a simple idea to make a board fly. Three embedded systems engineers envisioned a small machine with minimal components that would be suitable for indoor use. Today, Crazyflie is a palm-sized platform, making it ideal for education, research, and swarm robotics. [32, 33]

For a better understanding, the author suggests examining Figure 2.1, which shows a schematic representation of the CrazyFlie platform ecosystem. As can be seen, this ecosystem consists of three groups of devices and software. The first group includes the CrazyFlie 2.1 nano quadcopter, the Crazyflie Bolt 1.1 quadcopter controller, and the Roadrunner UWB positioning tag. Each device has advantages and disadvantages, but they are built on similar principles and can interact with components from other groups. The second group comprises the Python client, which runs on the user's local computer and communicates with devices through the Crazyradio PA radio transmitter. Finally, the third group consists of positioning systems, namely the Lighthouse Positioning System and the Loco Positioning System. [32, 33]

Figure 2.1 Crazyflie 2.1 without modules [34]



2.2 Crazyflie 2.1 quadcopter

In this bachelor's thesis, the author uses the Crazyflie 2.1 quadcopter. This drone is notable for its take-off weight of 27 grams and small dimensions of 92x92x29mm (motor-to-motor and including motor mount feet). In addition, its size and weight allow safe use in small indoor spaces, such as the Robotic Agents Laboratory (RoboAgeLab), which has a designated safe flight zone. [35]

Figure 2.2 Crazyflie 2.1 without modules [34]

Crazyflie 2.1 has a durable construction capable of withstand falls and collisions during experiments. Additionally, it is easy to assemble and requires no soldering, making it accessible to people without experience with embedded systems. The software is loaded wirelessly, offering convenience and speed in software development. The quadcopter also has an on-board charging feature. Another useful feature is real-time logging, graphing, and variable setting, in addition to the full use of expansion decks when using a Crazyradio or Crazyradio PA and a computer. [35]

The Crazyflie 2.1 board has a dual MCU architecture and features the following onboard microcontrollers:

1. STM32F405 main application MCU (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash);
2. nRF51822 radio and power management MCU (Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash);
3. micro-USB connector;
4. on-board LiPo charger with 100mA, 500mA, and 980mA modes available;
5. full-speed USB device interface;
6. partial USB OTG capability (USB OTG present but no 5V output);
7. 8KB EEPROM. [35]

Crazyflie 2.1 has a battery that provides a flight time of 7 minutes and takes approximately 40 minutes to recharge. The drone is able to receive radio signals at a frequency of 2.4GHz and a range of up to 1 kilometer using Crazyradio PA, broadening the spectrum of tasks it can perform. In addition, the drone can carry at least 15 grams of weight, allowing for the placement

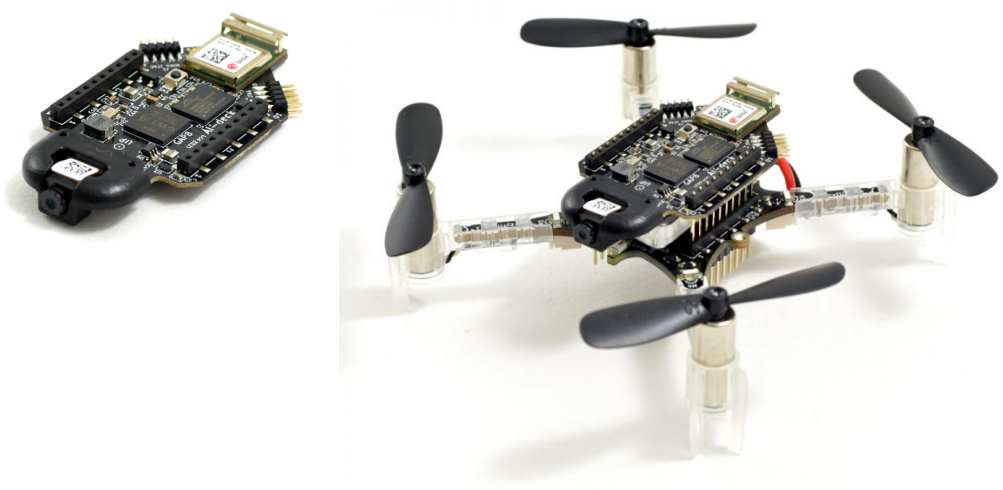
of additional modules such as the Lighthouse Positioning Deck, Loco Positioning Deck, Flow Deck V2, AI-Deck, and others. The author will use the Loco Positioning Deck and the AI-Deck in this thesis. [35]

2.3 AI-deck module

The AI-deck 1.1, designed for AI on the edge purposes, is built around the GAP8 RISC-V multi-core MCU. In addition, the QVGA monochrome camera and ESP32 WiFi MCU, a combination of which creates a pretty good platform to develop low-power AI on edge for a drone, are also available on the deck. The AI-deck 1.1 extends the computational capacity of the GAP8 and enables the implementation of complex workloads based on artificial intelligence that is driven onboard and can achieve fully autonomous navigation capabilities. [36]

Figure 2.3 AI-deck 1.1 and Crazyflie 2.1 with installed AI-deck module [37]

(a) AI-deck 1.1 module **(b)** Crazyflie 2.1 with installed AI-deck module



The ESP32 adds WiFi connectivity with the possibility to stream images and handle control. The developers of the AI-deck believe that this lightweight and low-power combination opens up many research and development areas for the microsized Crazyflie 2.X UAV. [36]

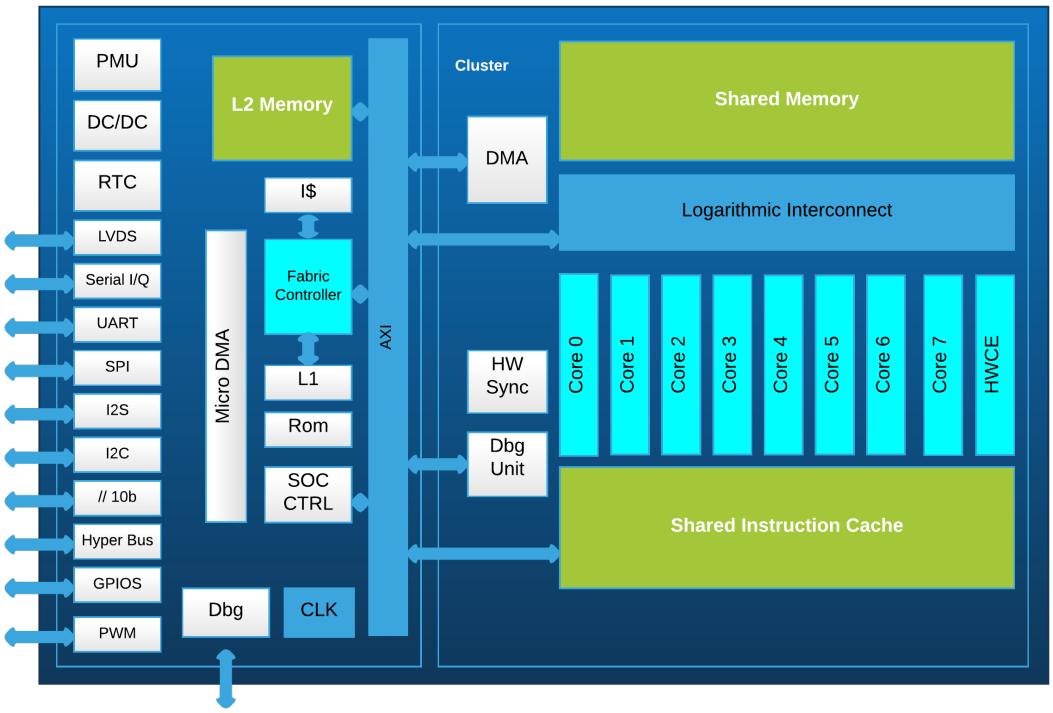
2.3.1 GAP8 processor

GAP8 is an IoT application processor that enables the massive implementation of low-cost battery-operated intelligent devices that capture, analyze, classify, and react to the combined flow of rich data sources such as images, sounds, radar signatures, and vibrations. GAP8 is specifically optimized to perform a wide range of image and audio algorithms, including convolutional neural network inference and signal processing, with extraordinary energy efficiency. GAP8 allows manufacturers of industrial and consumer products to integrate signal processing, artificial intelligence, and advanced classification into new classes of battery-operated wireless edge devices for IoT applications, including image recognition, person- and object-counting, machine health monitoring, home security, speech recognition, audio enhancement, consumer robotics and smart toys, as well as intelligent gadgets. By enabling autonomous operation, GAP8 drastically reduces the cost of deployment and operation of a wide range of smart edge devices. [38, 39]

The hierarchical and demand-driven architecture architecture of the GAP8 processor enables ultralow power operation by combining:

1. A collection of highly autonomous intelligent I/O peripherals for connection to cameras, microphones and other capture and control devices.
2. A fabric controller (FC) core for control,communications and security functions.
3. A compute cluster of 8 cores.
4. A Convolutional Neural Network accelerator(HWCE). [38, 39]

Figure 2.4 GAP8 Block Diagram [40]



The GAP8 processor has nine cores, eight working in parallel in a cluster for high-performance computations, and a single ”Fabric Controller” (FC) core for managing operations. The processor supports an extended RISC-V instruction set and has specialized instructions for optimizing targeted algorithms. It has a two-level memory structure: 512KB Level 2 memory accessible to all processors and DMA units and smaller Level 1 memory for the FC (16KB) and cluster cores (64KB). There is also access to external memory areas through HyperBus or quad-SPI peripherals. Internal memory is preferred over external memory access to optimize energy consumption and performance. The processor's capabilities quickly adapt to various applications' processing and energy requirements, making it suitable for tasks such as image processing, audio processing, and signal modulation. [38, 39]

2.3.2 Camera

The HM01B0 is an ultralow power CMOS Image Sensor that enables the integration of an "Always On" camera for computer vision applications such as gestures, intelligent ambient light and proximity sensing, tracking and object identification. The unique architecture of the sensor allows the sensor to consume a very low power of 2mW at QVGA 30FPS. [41, 42]

The HM01B0 sensor has the following parameters:

1. Active Pixel Array – 320 x 320
2. Pixel Size – 3.6 µm x 3.6 µm
3. Full Image Area – 1152 µm x 1152 µm
4. Diagonal (Optical Format) – 1.63 mm (1/11")
5. Color Filter Array – Monochrome and Bayer
6. Effective Focal Length – 0.66 mm [41, 42]

The HM01B0 sensor supports a window mode of 320 by 240, allowing for 60 frames per second. Additionally, a 2x2 binning can achieve a speed of 120 FPS. Monochrome video is transmitted through a 1, 2, or 8-bit interface. In order to optimize the power consumption, the HM01B0 sensor integrates a black-level calibration scheme, an automatic exposure and gain control loop, an auto-generator, and a motion detection circuit with an interrupt output to reduce host computations and sensor commands. Taking into account the parameters of this camera, it fits IoT, wearable devices, smart buildings, smartphones, tablets, and slim notebooks. [41, 42]

2.4 Crazyradio PA

The Crazyradio PA is a "*long-range open USB radio dongle based on the nRF24LU1+ from Nordic Semiconductor. It features a 20dBm power amplifier, LNA, and comes pre-programmed with Crazyflie compatible firmware.*" Crazyradio PA is depicted in figure 2.5 [43]. Key features include:

1. Radio power amplifier providing 20dBm output power
2. Over 1km line-of-sight range with Crazyflie 2.X
3. Low latency [43]

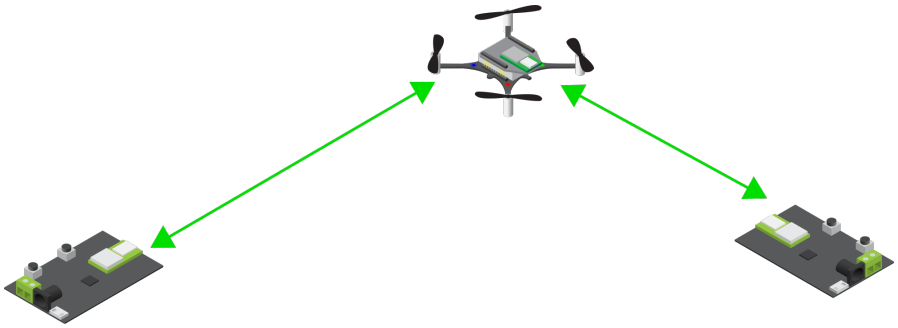
Figure 2.5 Crazyradio PA [44]

In this work, the Crazyradio PA is utilized to communicate with CrazyFlie 2.1 and to install and update the software on the AI-deck 1.1 module. The Crazyradio PA connects to the computer via USB, offering 125 radio channels and the capability to transmit information at speeds of up to 2Mbps. [43]

2.5 Loco positioning module

The Loco Positioning system is a local positioning system based on Ultra Wide Band radio, which determines the absolute 3D position of objects in space. In many ways, it is similar to a miniature GPS system. The example is shown in figure 2.6. [45, 46]

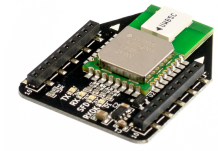
Figure 2.6 The Loco Positioning system [47]



The key components of the Loco Positioning system are Anchors and Tags. Anchors are placed within the indoor environment and serve as reference points, similar to satellites in GPS systems. Tags are placed on the object whose position we want to track. All information about the object's position is located on the tag, which, unlike other systems, does not require additional communication with other devices. As a result, Loco Positioning expands the autonomy capabilities of the Crazyflie 2.1 drone. In this thesis, the author uses a loco positioning set-up with eight anchors and one tag – loco positioning deck placed on the Crazyflie 2.1. The enchor setup is shown in Figure 2.8. [45, 46]

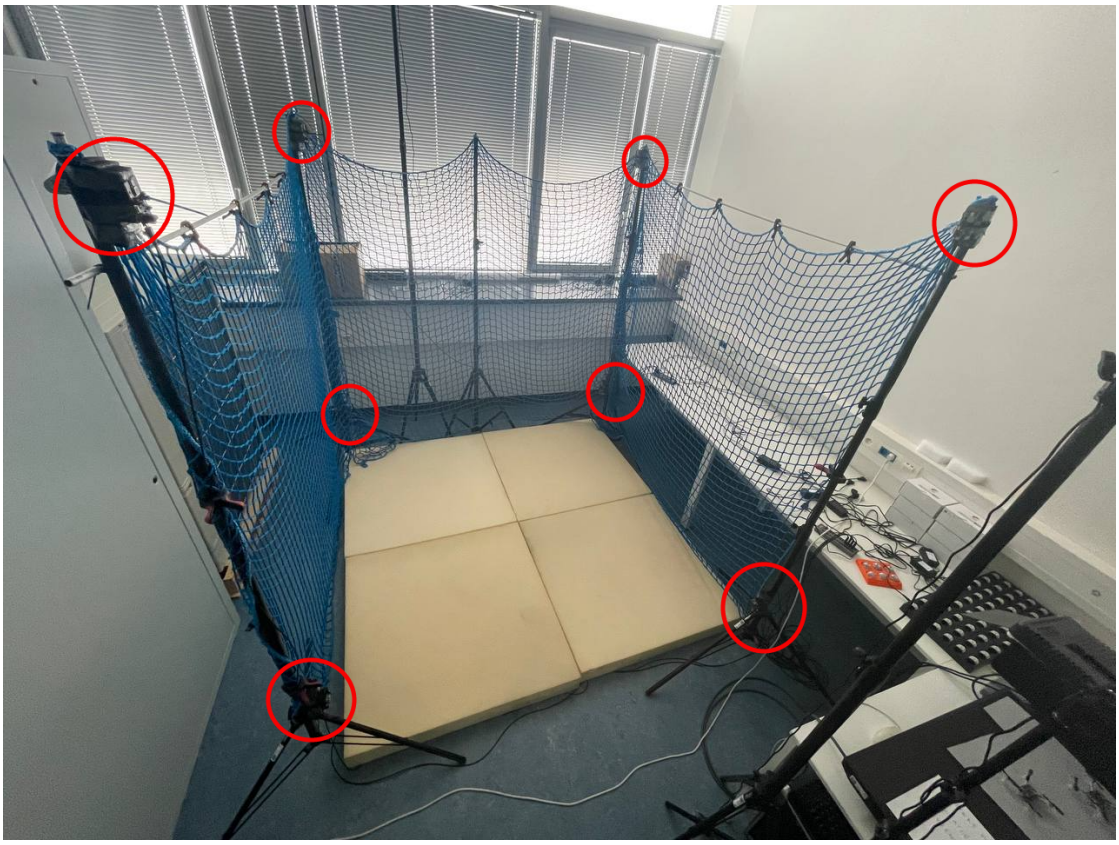
Figure 2.7 Loco positioning deck and Crazyflie 2.1 with installed AI-deck module [48]

(a) Loco positioning module



(b) Crazyflie 2.1 with installed Loco positioning module

Figure 2.8 8 anchor loco positioning setup from the lab. Anchors are outlined with red circles



Chapter 3

Implementation on Crazyflie

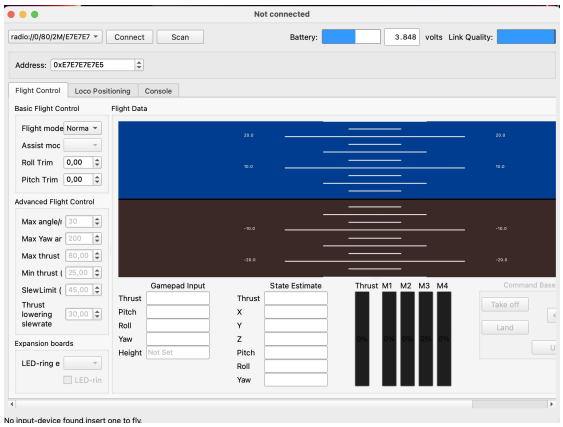
3.1 Work with Crazyflie quadcopter

During the implementation, the author created several building blocks that enable the drone to fly to the initial search point, after which the search process takes place. In the search process, the drone transmits images from its camera to the controlling computer, after which detection and tracking algorithms are applied to the images and measure the distance to the drone. The author will further describe each block separately in order to demonstrate and explain how the system, created using these blocks, successfully accomplishes the assigned tasks.

3.1.1 Hardware setup

The drone and its module configuration process begin with the installation of a specialized software called crazyflie-client (CC), developed by the same company that manufactures the drones. The developers have provided comprehensive documentation that describes the steps to install this program on all popular operating systems, namely Ubuntu, Windows, and Mac OS. The CC will be actively used in the software development and testing process. Figure 1 shows the graphical interface of CC.

Figure 3.1 Cfclient interface.



The next step in setting up the drone is connecting the AI-deck 1.1 module and installing the initial software. The process of installing the module is described in detail in the Crazyflie

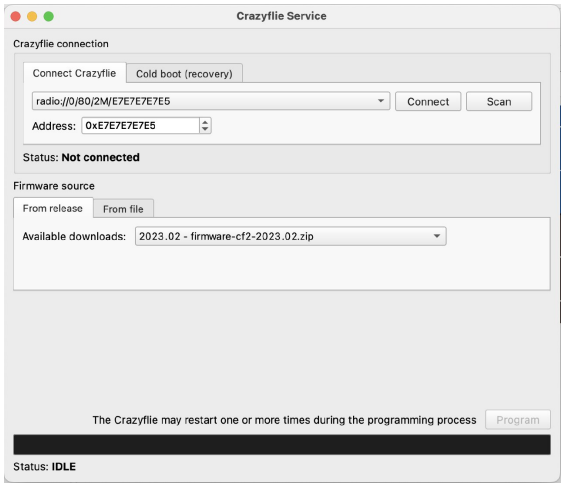
```
$ git clone https://github.com/bitcraze/aideck-gap8-bootloader.git
$ cd aideck-gap8-bootloader
$ docker run --rm -it -v $PWD:/module/ --device /dev/ttyUSB0 --privileged
-P bitcraze/aideck /bin/bash
-c 'export GAPY_OPENOCD_CABLE=interface/ftdi/olimex-arm-usb-tiny-h.cfg;
source /gap_sdk/configs/ai_deck.sh; cd /module/;  make all image flash'
```

Code listing 1 Insatalling a bootloader on AI-deck

documentation, is relatively straightforward, and does not require experience with hardware. Two modules can be connected to the Crazyflie simultaneously, one above and one below the drone's board. To install, insert the sockets located on the module board into the contacts on the drone board. Note that each module has a designated installation location. The AI-deck 1.1 had to be installed above the drone's board. After connecting the AI-deck 1.1, the subsequent step was to install the Crazyflie and AI-deck firmware. The following steps were required for installation:

1. Open the bootloader window shown in Figure 2, using 'Connect' → 'bootloader.'
2. Type the address of the Crazyflie quadcopter, press 'Scan,' and select Crazyflie's URI. Choose 'radio://...' (not 'usb://'). Press 'Connect.'
3. In the 'Firmware Source' section, select 2023.02 from 'Available downloads.'
4. Press "Program," after which the firmware will be installed.

Figure 3.2 Firmware installation interface.



Some versions of the AI-deck 1.1 need to flash the bootloader on the GAP8 separately; therefore, the module used in this work needed manual bootloader installation. This can only be done using the Olimex ARM-USB-TINY-H JTAG programmer and only from a native Linux computer or virtual machine. For this purpose, the author installed Ubuntu 22. The bootloader was then installed using Docker:

The bootloader installation using the Olimex ARM-USB-TINY-H JTAG programmer only needed to be performed once. Afterward, only over-the-air flashing was used for software loading.

3.1.2 Flying with loco positioning system

In the Bitcraze GitHub repository [49], libraries can be found to work with various modules. Among them, there is a library for the loco positioning module. The main component of this library is the PositionHlCommander class, which is responsible for high-level drone control using the loco positioning module. This class has the following methods:

1. take_off – responsible for the drone taking off from the starting position.
2. land – responsible for returning the drone to the initial position.
3. left, right, forward, back – responsible for moving a specific distance to the left, right, forward, and backward relative to the drone's initial orientation.
4. up, down – responsible for ascent and descent moving.
5. move_distance – responsible for moving the drone a specified distance in three-dimensional space.
6. go_to – responsible for moving the drone to a specified position in three-dimensional space.

A limitation of this class is that all movements occur relative to the drone's initial orientation(yaw), which does not change during the flight. To achieve the maximum potential of the loco positioning system, the author created the PositionExtendedCommander class, which is responsible for the drone's orientation during movement. In the new version, the author added the `__rotate` method, which is responsible for changing the drone's orientation, as well as the left, right, forward and back methods, which use sine and cosine geometry to calculate the drone's new position. The working principle can be illustrated using the left method:

```
def left(self, distance_m, velocity=DEFAULT):
    """
    Go left

    :param distance_m: The distance to travel (meters)
    :param velocity: The velocity of the motion (meters/second)
    :return:
    """
    perpendicular_angle = self._yaw + math.pi / 2
    x_component = math.cos(perpendicular_angle) * distance_m
    y_component = math.sin(perpendicular_angle) * distance_m
    self.move_distance(x_component, y_component, 0.0, velocity)
```

Code listing 2 Moving to the left respecting current yaw

The `self._yaw` variable represents the drone's orientation at the beginning of the movement, while the variables `x_component` and `y_component` store the change in the position of the drone along the x and y axes, respectively. This extension allows changing the drone's orientation at any point during the flight. It should be noted that communication between the drone and the computer running the flight program is carried out using the Crazyradio PA radio communicator.

In general, the flight process proceeds as follows. Upon the start of the script, the communication channel between the drone and the computer is established. After a successful connection, the drone assumes its initial position. The drone then begins processing movement commands. Once the drone stops receiving movement commands or the script reaches its end, it returns to its initial position and lands. Using the `sleep()` function from the `time` library or using loops, the

drone can remain in its current position. The code for the entire PositionExtendedCommander class can be found in Appendix F.

3.1.3 Images streaming with AI-deck

The streaming of images from the AI-deck camera to the computer involves two components. The first is a streamer program that runs on the GAP8 processor and transmits the image using the AI-deck's Wi-Fi module. The second component is a Python script that connects to the streamer via sockets to receive the images.

The author found a ready-to-use example of an image streamer in a Bitcraze GitHub repository [49]. Image streaming can occur in two modes: RAW or JPEG. To configure the mode, the code is in the wifi-img-streamer.c file needs to be modified:

```
$ git clone https://github.com/bitcraze/aideck-gap8-examples.git
```

Code listing 3 Cloning a AI-deck GAP8 examples repository

```
typedef enum
{
  RAW_ENCODING = 0,
  JPEG_ENCODING = 1
} __attribute__((packed)) StreamerMode_t;

static StreamerMode_t streamerMode = RAW_ENCODING;
```

Code listing 4 Modifying a wifi-img-streamer.c file

After the initial configuration, it is necessary to build and flash the streamer code as follows:

```
$ cd aideck-gap8-examples
$ docker run --rm -v ${PWD}:/module aideck-with-autotiler tools/build/make-example
examples/other/wifi-img-streamer image
$ cfloader flash examples/other/wifi-img-streamer/BUILD/GAP8_V2/GCC_RISCV_FREERTOS/
target.board.devices.flash.img deck-bcAI:gap8-fw -w radio://0/80/2M/E7E7E7E7E5
```

Code listing 5 Building and flushing images streamer

The author modified an existing Python script in the Bitcraze GitHub repository [49] to receive the images. As a result of the modifications, the images_streamer.py script was created. Its working principle can be described as follows:

1. Import required libraries: argparse, time, socket, os, struct, numpy.
2. Define command line arguments for setting the AI-deck IP address and port, with default values provided.
3. Establish a connection to the AI-deck using a socket and display a message indicating successful connection.

4. Define a function, rx_bytes, to receive a specified number of bytes from the connected socket.

5. In the while loop, the code continuously receives and processes images by unpacking packet and image headers, reassembling the image stream, updating the image count and performance metrics, and converting the image stream into a NumPy array with the correct dimensions.

   The code of the entire image streaming script can be found in Appendix C.

3.2 Object detection

3.2.1 Data collection and annotation

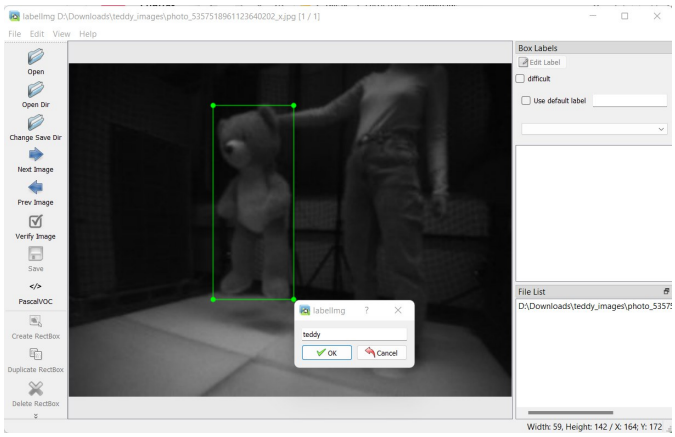
Initially, the author used a balloon as the object for detection. However, it was discovered that the balloon was not effectively detected under low image quality and laboratory lighting conditions. Additionally, the balloon lacked distinctive features necessary for detection.

As a result, the author opted to use a stuffed teddy bear due to its numerous features suitable for detection in any lighting condition. The model was trained using 1164 images of the teddy bear captured under laboratory conditions, at various distances and angles. Four photographs of the teddy bear are shown in Figure 3.3.

Figure 3.3 Samples of training images

During the training process, it was necessary to annotate the object in the photographs. The "labelImg" program was used to simplify this task by creating annotations and saving them as XML files in PASCAL VOC format. Figure 3.4 displays the interface of the "labelImg" program.

Figure 3.4 labelImg interface



```
batch_size = 16
num_steps = 50000
num_eval_steps = 1000
```

Code listing 6 Setting a hyperparameters

As a result of the annotation, XML files were created. One of the created files will be shown in the Appendix A. The complete code of data processing is given in Appendix B

3.2.2 Model training

To train and test the SSD MobileNet V2 FPNLite 320x320 model, the author utilized the Google Colab online service. This decision was made due to its ability to utilize external computational resources, significantly speeding up the model training process. In addition, Colab offers an environment where Jupyter notebooks can be executed.

The author created a Jupyter notebook called "teddy_ssd_mobilenet_v2_fnlite.ipynb," which contains step-by-step instructions for training, testing, and exporting the object detection model for a teddy bear. In the following text, the author will outline several crucial steps of the created notebook. The code of the entire "teddy_ssd_mobilenet_v2_fnlite.ipynb" notebook can be found in Appendix B.

The author used and modified the existing Jupyter notebook [50]. The author used the TensorFlow2 library to train the model. To begin with, they installed all the required libraries and dependencies at the beginning of the notebook. Subsequently, training data was prepared by generating CSV files that included details about the images, such as filename, width, height, class, xmin, ymin, xmax, and ymax. These CSV files were converted to tf record files, an effective file format for storing large volumes of data for machine learning purposes. Lastly, the author fine-tuned the hyperparameters as follows:

1. batch_size – The batch size is the number of training samples used in each update of the model weights during training. A smaller batch size means that the model will update its weights more frequently, while a larger batch size will result in less frequent updates. The size of the batch can affect the speed of training and the stability of convergence. In this case, the batch size is set to 16, which means that the model will use 16 samples in each update.

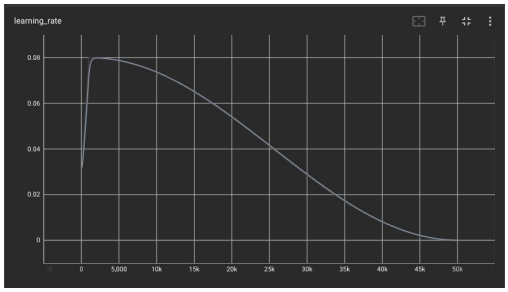
2. num_steps – This hyperparameter specifies the total number of steps (or iterations) that the training process will take. Each step consists of processing one batch of data and updating the model weights accordingly. In this case, the model will be trained for 50,000 steps before stopping. Keep in mind that this is different from epochs, which refer to the number of times the model goes through the entire training dataset.

3. num_eval_steps – This hyperparameter specifies the number of steps used to evaluate the model during the training process. Evaluation is typically done on a separate dataset, known as the validation or evaluation dataset, to measure the model's performance on unseen data. In this case, the model will be evaluated every 1,000 steps. This allows you to monitor the model's performance during training and identify potential issues such as overfitting.

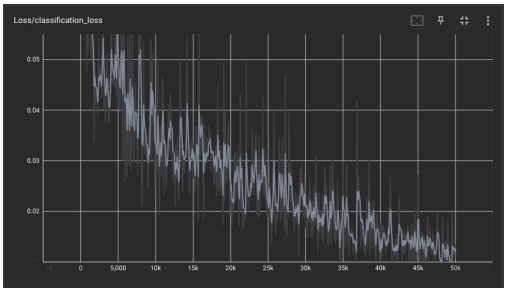
After adjusting the hyperparameters, the script is executed to train the model with the input provided. In the next section, the author will present and clarify the training results using graphs obtained during the training process.

The learning rate is a key element in training a model that decides how much the model's weights are updated. A higher learning rate means that the model will make more significant

adjustments, whereas a lower learning rate will result in more minor adjustments. Figure 3.5a shows a graph that illustrates the data from the training process, where the learning rate starts at 0.08 and decreases to 0 after 50,000 steps. This approach is commonly referred to as learning rate scheduling, which helps the model to converge to the best possible solution by gradually decreasing the learning rate over time.



(a) Learning rate graph

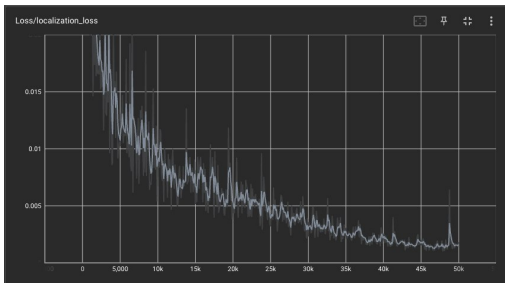


(b) Classification loss graph

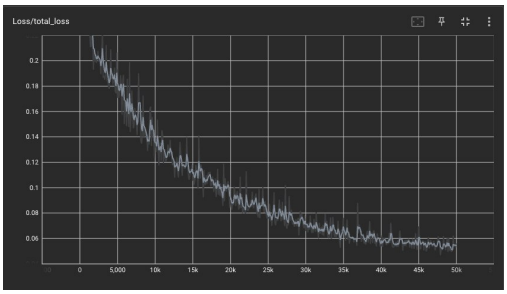
Figure 3.5 Training graphs

The classification loss measures the accuracy of the model in predicting the class labels of the input samples. The graph in Figure 3.5b shows that the classification loss decreases from 0.11 to 0.0016 during training, indicating that the model's ability to classify the data has improved greatly.

When predicting the location of objects in input data, there is an associated error called localization loss. This is especially important in object detection and segmentation tasks, where the model must determine the object's class label and position in the input image. The graph in Figure 3.6a shows that the loss of localization starts at 0.11 and decreases to 0.0018 throughout the training process, indicating an improvement in the model's ability to locate objects.



(a) Learning rate graph



(b) Classification loss graph

Figure 3.6 Training graphs

To avoid overfitting, regularization loss is incorporated into the loss function to penalize too complex models. This involves adding a penalty term based on the model's weights. As shown in Figure 3.6b, the regularization loss graph starts at 0.15 and gradually decreases to 0.04 during training, indicating a decrease in the complexity of the model.

The total loss combines the classification, localization, and regularization loss. The primary goal of the training process is to reduce this overall goal. The graph in Figure 1 shows the total loss decreasing from 0.396 to 0.054, which means that the model has successfully learned to reduce the error in the training data.

The author gained valuable insights into the training process by analyzing these graphs. They show how the model's performance in different areas, such as classification, localization, and

complexity, changes over time as it learns from the training data. The fact that the classification loss, localization loss, and total loss values decrease indicates that the model is effectively learning from the data. However, the decreasing learning rate and loss of regularization suggest that model complexity is being regulated to avoid overfitting.

3.3 Distance estimation

After obtaining the bounding box as a result of object detection, the estimation of the distance between the object and the camera is calculated. The calculation is done through the implementation of the following code:

```
def distance_to_object(
        bbox_height_in_pixels,
        real_object_height=88,
        effective_focal_length=0.66,
        pixel_size=3.6
    ):
    """
    Calculate the distance to the object from the camera.

    :param image_height_in_pixels: Height of the object in the image in pixels.
    :param real_object_height: Real height of the object in centimeters.
           Default is 88 cm.
    :param effective_focal_length: Effective focal length of the camera in millimeters.
           Default is 0.66 mm.
    :param pixel_size: Size of a pixel in micrometers. Default is 3.6 µm.

    :return: Distance to the object from the camera in centimeters.
    """
    # Convert pixel size to millimeters
    pixel_size_mm = pixel_size / 1000.0

    # Convert real_object_height to millimeters
    real_object_height_mm = real_object_height * 10

    # Calculate distance to object
    distance =
        (real_object_height_mm * effective_focal_length)
        / (image_height_in_pixels * pixel_size_mm)

    # Convert distance to centimeters
    distance_cm = distance / 10

    return distance_cm
```

Code listing 7 Distance estimation

3.4 Object tracking

This document describes the steps involved in object tracking using the Kalman Filter and the choice of parameters used in the given code. The code provided is implemented in Python and utilizes the OpenCV library.

The object tracking process using the Kalman filter consists of the following steps:

1. Initialize the Kalman Filter with appropriate parameters.
2. Convert the initial bounding box of the object to a state vector.
3. Predict the next state using the Kalman Filter.
4. Update the state with the new measurement.
5. Convert the current state to a bounding box.

The transition matrix, denoted as A, is an 8×8 matrix that models the relationship between the current state and the next state. The matrix A contains values that describe the relationships between the state variables, which include the center coordinates of the bounding box (x, y), the width and height (w, h), and their respective velocities (v_x, v_y, v_w, v_h). The diagonal elements of the matrix are set to one, indicating that the state variables are directly carried over from one time step to the next. Meanwhile, the off-diagonal elements in the first four rows represent how velocities contribute to the position and size of the bounding box. A is defined as follows:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

The measurement matrix, denoted H, is a 4×8 matrix that maps the state vector to the measurement vector. The matrix H shows how the observed variables relate to the state variables. The variables observed in this case are the center coordinates of the bounding box (x, y), as well as the width and height (w, h). The diagonal elements of the matrix have a value of one, indicating that the observed variables correspond directly to the state variables. On the other hand, zero elements in the matrix mean that velocities (v_x, v_y, v_w, v_h) do not directly affect the observed variables. H is defined as follows:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.2}$$

The process noise covariance matrix, denoted as Q, is an 8×8 matrix that models the uncertainty in the state transition. The choice of the process noise covariance matrix in this code is as follows:

$$Q = 5 \cdot I_8 \tag{3.3}$$

The measurement noise covariance matrix, denoted as R, is a 4×4 matrix that models the uncertainty in the measurements. The choice of the measurement noise covariance matrix in this code is as follows:

$$R = 1 \times 10^{-5} \cdot I_4 \tag{3.4}$$

Where I_4 and I_8 are identity matrices of size 4×4 and 8×8, respectively. Identity matrices are square matrices with ones on the diagonal and zeros elsewhere. They have the property that, when multiplied by another matrix, they do not change the other matrix.

In this context, I_4 and I_8 are defined as follows:

$$I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad I_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

I_8 and I_4 are used as a base for creating the process noise covariance matrix Q and the measurement noise covariance matrix R.

The values chosen for the process noise covariance matrix and measurement noise covariance matrix have a significant impact on how well the Kalman filter can track an object. These parameters can be adjusted to balance the filter's responsiveness to state changes and noise reduction in the state estimates. For instance, increasing the process noise covariance values can make the filter more responsive to sudden changes in the object's state, but it may also introduce more noise in the estimates. Conversely, decreasing the measurement noise covariance values can increase the filter's reliance on measurements, which could lead to overfitting the observed data and reduced smoothness in the state estimates. The code of the Kalman filter implementation can be found in Appendix D.

3.5 Final solution and tests

The individual building blocks were combined into a single script called flying_streamer.py, which can be found in Appendix G. In the following text, the author will explain how this script works.

Initially, all necessary libraries and dependencies were imported. Subsequently, the object detection model was initialized. Then, the connection to the drone and the AI-deck was set up. Subsequently, the drone was given the command to take off and move to the starting point to start an object search. Once the drone reached the designated position, the object detection process was started. Upon successful detection, object tracking and distance estimation were performed. It is worth noting that if the object was not detected in the last ten frames, the drone would change its orientation and attempt to detect the object again.

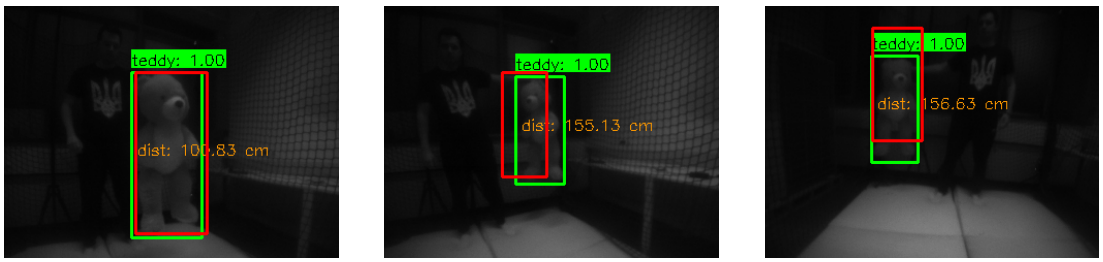


Figure 3.7 Result images

As a result of the testing under laboratory conditions, images were obtained, which can be seen in Figure 3.7. Upon successful detection, the object is displayed within a green bounding box. Above the bounding box, the detected class name is displayed; in our case, it is "teddy", along with the detection accuracy. A red rectangle represents the object tracking prediction. Finally, the distance estimation is printed inside the bounding box.

The testing provided helpful information to analyze the precision and correctness of detection, tracking, and distance estimation from the object to the drone. The author will now describe and analyze each of the graphs created from the obtained data.

Figure 3.8 Detection success rate graph

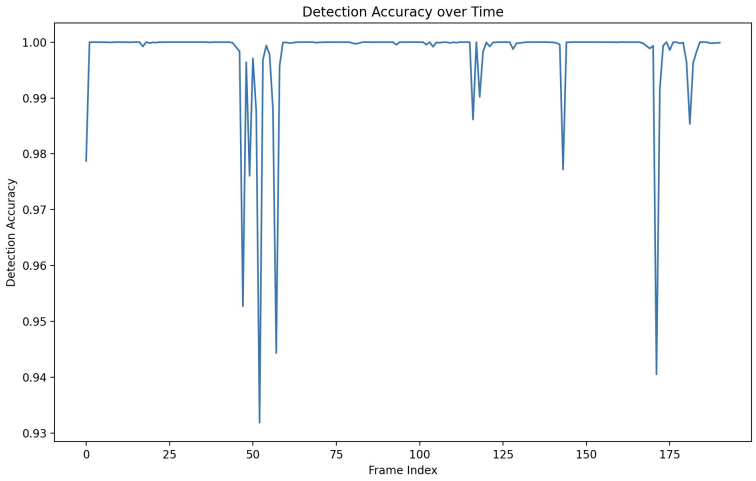


Figure 3.8 shows the object detection success rate graph relative to the number of frames at a given moment. For example, the graph will display the percentage of successful detection over the previous ten frames at the point for the tenth frame. This graph shows that the object was detected in most frames. However, it also highlights segments where the detection rate drops sharply, indicating that the object was out of the drone's field of view during those segments.

Figure 3.9 Detection accuracy graph

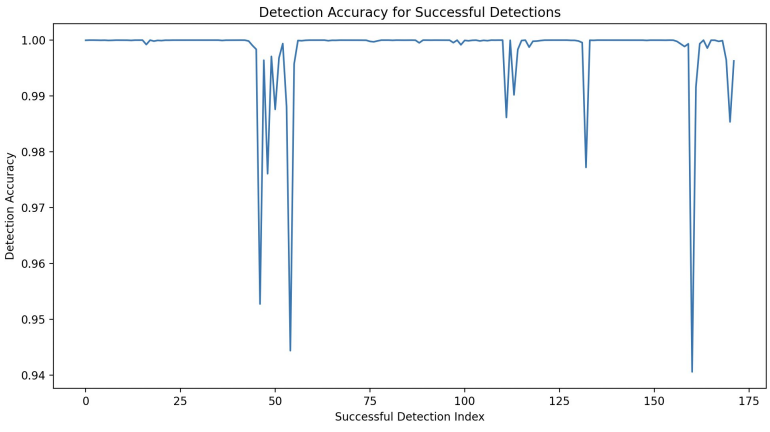
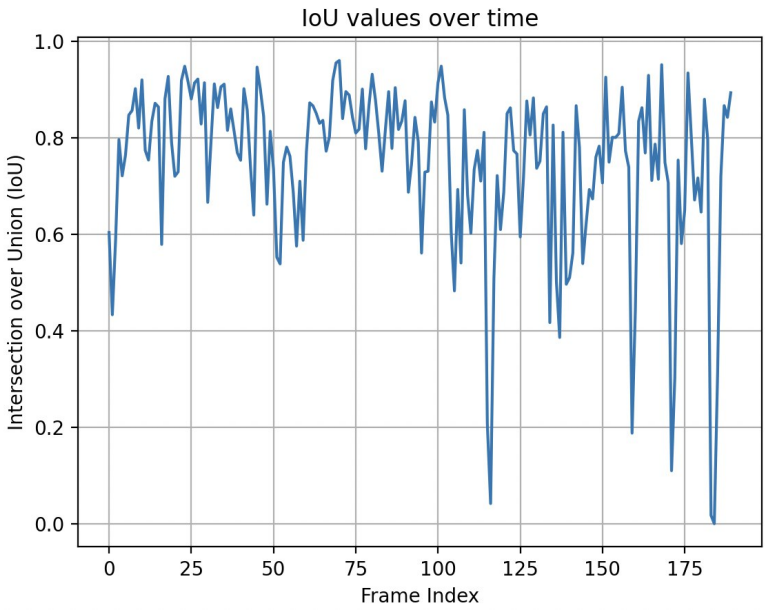


Figure 3.9 presents the object detection accuracy graph. Upon analyzing this graph, the author concludes that the object detection accuracy has excellent accuracy and very low variation.

Figure 3.10 displays the tracking algorithm's success rate graph based on the Intersection

Figure 3.10 Tracking accuracy graph



over Union (IOU) metric. IOU measures the degree of overlap between the bounding box created during detection and the predicted bounding box. The values of this metric range from 0 to 1. Low values indicate that the object position prediction algorithm is inaccurate, while high values indicate high accuracy. Figure 3.10 shows that the prediction accuracy varies between 0.5 and 0.9, which the author believes is a good result. It is worth noting that the shallow values on the graph are associated with a sudden change in the object's motion. After a low value, there is always a significant increase, indicating that the algorithm quickly adapts.

Figure 3.11 Distance estimation accuracy graph

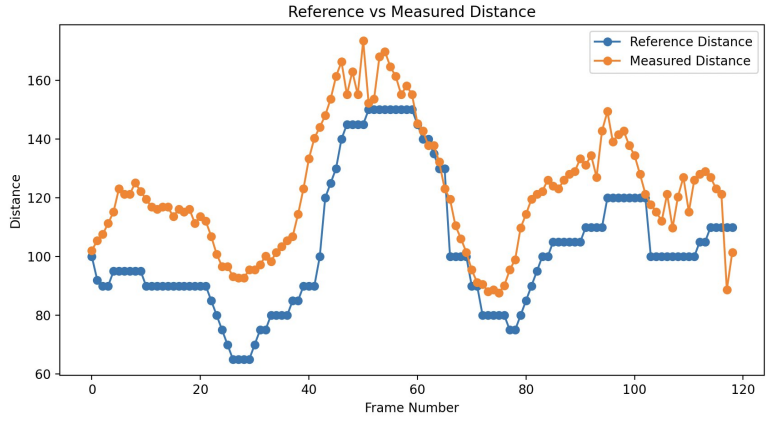


Figure 3.11 shows two graphs – the first represents the actual distance of the object from the camera, and the second represents the distance calculated by the algorithm. It should be noted that the actual distance may have errors due to drone flight instability. After analyzing the graphs, the author concludes that there is a systematic error in distance measurement, leading to nearly 20 percent inaccuracy. In the author's opinion, this error arises from a combination of

factors, such as inaccuracies in bounding box dimensions and the drone's motion.

The average values of detection, detection accuracy, IOU and distance estimation are presented in Table 3.1. The data in the table demonstrate that, overall, the object detection, tracking, and distance estimation algorithms perform with good accuracy.

Metric	**Average Value in %**
Detection	87.21
Detection Accuracy	99.78
Intersection over Union (IOU)	74.85
Average distance estimation error	19.62

Table 3.1 Average values of detection, detection accuracy, IOU and distance estimation

Conclusion

The goal of the theoretical part of this thesis was to study existing methods for visual object detection and tracking. The first goal of the realization part was to implement new or modify existing visual detection and object tracking algorithms on the Crazyflie quadcopter using the AI-deck module with an onboard camera and the stationary localization system. The second goal was to perform relevant tests on the Crazyflie quadcopter in the Robotic Agents Laboratory.

All the set goals for the theoretical part were fully achieved as contemporary and older methods for visual object detection and tracking were investigated. Likewise, the goals of the practical part were also fully accomplished, as the object was successfully detected and the tracking algorithm functioned correctly during testing. Furthermore, the author was able to investigate and implement distance estimation from the object to the drone using a single camera.

During the process of working on this thesis, the author gained invaluable experience working with computer vision methods and the Crazyflie platform. Considering the acquired experience, the author sees great potential in further work with the Crazyflie drone, specifically in extending the drone's autonomy and improving the accuracy in performing object detection and tracking tasks.

Bibliography

1. ZOU, Zhengxia; CHEN, Keyan; SHI, Zhenwei; GUO, Yuhong; YE, Jieping. Object Detection in 20 Years: A Survey. *Proceedings of the IEEE* [online]. 2023, vol. 111, no. 3, pp. 257–276. Available from DOI: 10.1109/JPROC.2023.3238524.
2. MOHSAN, Syed Agha Hassnain; OTHMAN, Nawaf Qasem Hamood; LI, Yanlong; ALSHARIF, Mohammed H.; KHAN, Muhammad Asghar. Unmanned aerial vehicles (UAVs): practical aspects, applications, open challenges, security issues, and future trends. *Intelligent Service Robotics* [online]. 2023, vol. 16, no. 1, pp. 109–137 [visited on 2023-05-05]. ISSN 1861-2784. Available from DOI: 10.1007/s11370-022-00452-4.
3. ZHAO, Zhong-Qiu; ZHENG, Peng; XU, Shou-Tao; WU, Xindong. Object Detection With Deep Learning: A Review. *IEEE Transactions on Neural Networks and Learning Systems* [online]. 2019, vol. 30, no. 11, pp. 3212–3232. Available from DOI: 10.1109/TNNLS.2018.2876865.
4. ZOU, Xinrui. A Review of Object Detection Techniques. In: *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)* [online]. 2019, pp. 251–254. Available from DOI: 10.1109/ICSGEA.2019.00065.
5. VIOLA, P.; JONES, M. Rapid object detection using a boosted cascade of simple features. In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001* [online]. 2001, vol. 1, pp. I–I. Available from DOI: 10.1109/CVPR.2001.990517.
6. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/document/990517/figures.
7. DALAL, N.; TRIGGS, B. Histograms of oriented gradients for human detection. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* [online]. 2005, vol. 1, 886–893 vol. 1. Available from DOI: 10.1109/CVPR.2005.177.
8. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/document/1467360/figures%5C#figures.
9. FELZENSZWALB, Pedro; MCALLESTER, David; RAMANAN, Deva. A discriminatively trained, multiscale, deformable part model. In: *2008 IEEE Conference on Computer Vision and Pattern Recognition* [online]. 2008, pp. 1–8. Available from DOI: 10.1109/CVPR.2008.4587597.
10. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/document/4587597/figures%5C#figures.

11. ZHAO, Zhong-Qiu; ZHENG, Peng; XU, Shou-Tao; WU, Xindong. Object Detection With Deep Learning: A Review. *IEEE Transactions on Neural Networks and Learning Systems* [online]. 2019, vol. 30, no. 11, pp. 3212–3232. Available from DOI: 10.1109/TNNLS.2018.2876865.
12. TAMMINA, Srikanth. Transfer learning using vgg-16 with deep convolutional neural network for classifying images. *International Journal of Scientific and Research Publications (IJSRP)* [online]. 2019, vol. 9, no. 10, pp. 143–150.
13. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/abstract/document/8627998/figures%5C#figures.
14. LIN, Tsung-Yi; GOYAL, Priya; GIRSHICK, Ross; HE, Kaiming; DOLLÁR, Piotr. Focal Loss for Dense Object Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* [online]. 2020, vol. 42, no. 2, pp. 318–327. Available from DOI: 10.1109/TPAMI.2018.2858826.
15. YILMAZ, Alper; JAVED, Omar; SHAH, Mubarak. Object Tracking: A Survey. *ACM Comput. Surv.* [Online]. 2006, vol. 38, no. 4, 13–es. ISSN 0360-0300. Available from DOI: 10.1145/1177352.1177355.
16. ALI, Ahmad; JALIL, Abdul; NIU, Jianwei; ZHAO, Xiaoke; RATHORE, Saima; AHMED, Javed; IFTIKHAR, Muhammad Aksam. Visual object tracking—classical and contemporary approaches. *Frontiers of Computer Science* [online]. 2016, vol. 10, no. 1, pp. 167–188. ISSN 2095-2236. Available from DOI: 10.1007/s11704-015-4246-3.
17. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/document/8964761/figures%5C#figures.
18. SOLEIMANITALEB, Zahra; KEYVANRAD, Mohammad Ali. Single Object Tracking: A Survey of Methods, Datasets, and Evaluation Metrics. *arXiv preprint arXiv:2201.13066* [online]. 2022.
19. SOLEIMANITALEB, Zahra; KEYVANRAD, Mohammad Ali; JAFARI, Ali. Object Tracking Methods:A Review. In: *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)* [online]. 2019, pp. 282–288. Available from DOI: 10.1109/ICCKE48569.2019.8964761.
20. GAT, Itay; BENADY, Meny; SHASHUA, Amnon. *A Monocular Vision Advance Warning System for the Automotive Aftermarket* [online]. 2005. [visited on 2023-05-05]. ISBN 0148-7191.
21. SEO, Beom-Su; PARK, Byungjae; CHOI, Hoon. Sensing Range Extension for Short-Baseline Stereo Camera Using Monocular Depth Estimation. *Sensors (Basel, Switzerland)* [online]. 2022, vol. 22, no. 12, p. 4605. ISBN 1424-8220.
22. THAHER, Raad H.; HUSSEIN, Zaid K. Stereo Vision Distance Estimation Employing SAD with Canny Edge Detector. *International journal of computer applications* [online]. 2014, vol. 107, no. 3, pp. 38–43. ISBN 0975-8887.
23. GMBH, ResearchGate. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.researchgate.net/figure/Fig-1-the-positions-of-two-cameras-and-their-image-planes-of-projection-Figure-1_fig1_284488453.
24. ABDUL, Haseeb Muhammad; DANIJELA, Ristić-Durrant; AXEL, Gräser; MILAN, Banić; DUŠAN, Stamenković. Multi-DisNet: Machine Learning-Based Object Distance Estimation from Multiple Cameras. In: TZOVARAS, Dimitrios; GIAKOUMIS, Dimitrios; VINCZE, Markus; ARGYROS, Antonis (eds.). *Computer Vision Systems* [online]. Cham: Springer International Publishing, 2019, pp. 457–469. ISBN 978-3-030-34995-0.
25. GMBH, ResearchGate. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.researchgate.net/figure/Distance-estimation-of-the-lane-width-at-the-point-where-the-vehicle-meets-the-road_fig4_261112107.

26. GAT, Itay; BENADY, Meny; SHASHUA, Amnon. A Monocular Vision Advance Warning System for the Automotive Aftermarket. *SAE Transactions* [online]. 2005, vol. 114, pp. 403–410 [visited on 2023-04-19]. ISSN 0096736X, ISSN 25771531. Available from: http://www.jstor.org/stable/44682447.
27. AG, Springer Nature Switzerland. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://link.springer.com/content/pdf/10.1007/s11704-015-4246-3.pdf?pdf=button.
28. MENG, Jing; JIANG, Ping; WANG, Jianmin; WANG, Kai. A MobileNet-SSD Model with FPN for Waste Detection. *Journal of Electrical Engineering & Technology* [online]. 2022, vol. 17, no. 2, pp. 1425–1431. ISSN 2093-7423. Available from DOI: 10.1007/s42835-021-00960-w.
29. GOSYSTECH. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.goozit.com/tutorial/ObjectDetection.
30. SANDLER, Mark; HOWARD, Andrew; ZHU, Menglong; ZHMOGINOV, Andrey; CHEN, Liang-Chieh. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* [online]. 2018, pp. 4510–4520. Available from DOI: 10.1109/CVPR.2018.00474.
31. IEEEXPLORE.IEEE.ORG. *Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://ieeexplore.ieee.org/document/8578572/figures%5C#figures.
32. AB, Bitcraze. *System overview* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/system/.
33. AB, Bitcraze. *Bitcraze* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/about/bitcraze/.
34. AB, Bitcraze. *Bitcraze Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/products/crazyflie-2-1/.
35. AB, Bitcraze. *Datasheet Crazyflie 2.1* [online]. 2021. Version 3 [visited on 2023-04-30]. Available from: https://www.bitcraze.io/documentation/hardware/crazyflie_2_1/crazyflie_2_1-datasheet.pdf.
36. AB, Bitcraze. *Datasheet AI-deck 1.1* [online]. 2022. Revision 2 [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/hardware/ai_deck_1_1/ai_deck_1_1-datasheet.pdf.
37. AB, Bitcraze. *Bitcraze Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://store.bitcraze.io/products/ai-deck-1-1.
38. TECHNOLOGIES, GreenWaves. *GAP8 IoT Application Processor* [online]. 2020. [visited on 2023-05-05]. Available from: https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1_9.pdf. Version 1.9.
39. TECHNOLOGIES, GreenWaves. *GAP8 Manual* [online]. 2020. [visited on 2023-05-05]. Available from: https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html. Generated on Tue Dec 1 2020 15:49:10.
40. TECHNOLOGIES, GreenWaves. *GAP8 manual figures* [online]. 2020. [visited on 2023-05-05]. Available from: https://greenwaves-technologies.com/manuals/BUILD/HOME/html/index.html. Generated on Tue Dec 1 2020 15:49:10.
41. HIMAX TECHNOLOGIES, Inc. *HM01B0 Ultralow Power CIS* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.himax.com.tw/products/cmos-image-sensor/always-on-vision-sensors/hm01b0/.
42. HIMAX TECHNOLOGIES, Inc. *HM01B0-MNA-01FT870* [online]. 2019. Version 1 [visited on 2023-05-05]. Available from: https://cdn.sparkfun.com/assets/7/f/c/8/3/HM01B0-MNA-Datasheet.pdf.

43. AB, Bitcraze. *Datasheet Crazyradio PA 2.4 GHz USB dongle* [online]. 2022. Revision 3 [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/hardware/crazyradio_pa/crazyradio_pa-datasheet.pdf.
44. AB, Bitcraze. *Bitcraze Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/products/crazyradio-pa/.
45. AB, Bitcraze. *Loco Positioning System* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/system/positioning/loco-positioning-system/.
46. AB, Bitcraze. *Datasheet Loco Positioning Deck* [online]. 2020. Revision 1 [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/hardware/loco_deck/loco_deck-datasheet.pdf.
47. AB, Bitcraze. *Bitcraze Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/documentation/system/positioning/loco-positioning-system/.
48. AB, Bitcraze. *Bitcraze Figures* [online]. 2023. [visited on 2023-05-05]. Available from: https://www.bitcraze.io/products/loco-positioning-deck/.
49. AB, Bitcraze. *aideck-gap8-examples* [online]. 2022. [visited on 2023-05-05]. Available from: https://github.com/bitcraze/aideck-gap8-examples.
50. MEHTA, Vidish. *ObjectDetection* [online]. 2021. [visited on 2023-05-05]. Available from: https://github.com/VidishMehta001/ObjectDetection. commit 0f16e54.

Appendix A

XML annotation

```
<annotation>
        <folder>teddy_images</folder>
        <filename>photo_5357518961123640202_x.jpg</filename>
        <path>D:\Downloads\teddy_images\photo_5357518961123640202_x.jpg</path>
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>324</width>
                <height>244</height>
                <depth>1</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>teddy</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>105</xmin>
                        <ymin>30</ymin>
                        <xmax>164</xmax>
                        <ymax>172</ymax>
                </bndbox>
        </object>
</annotation>
```

Code listing 8 Annotation in PASCAL VOC format

Appendix B

teddy_ssd_mobilenet_v2_fnlite.ipynb

teddy_ssd_mobilenet_v2_fnlite

April 29, 2023

0.1 Step 1: Mount Google Drive

[ ]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

[ ]:
```
## Run this if you want to access models from google drive
import os
print(os.getcwd())
!ls
```

0.2 Step 2: Install Dependencies

[ ]:
```
!pip install -U --pre tensorflow=="2.12.0"
```

[ ]:
```
import os
import pathlib
os.chdir('/content/gdrive/My Drive/')
print(os.getcwd())

# Clone the tensorflow models repository if it doesn't already exist
if "models_teddy" in pathlib.Path.cwd().parts:
  while "models_teddy" in pathlib.Path.cwd().parts:
    os.chdir('..')
elif not pathlib.Path('models_teddy').exists():
  !git clone --depth 1 https://github.com/tensorflow/models models_teddy
```

[3]:
```
os.chdir('/content/gdrive/My Drive/')
```

[ ]:
```
# Object detection API
%%bash
cd models_teddy/research/
protoc object_detection/protos/*.proto --python_out=.
cp object_detection/packages/tf2/setup.py .
python -m pip install .
```

[ ]:
```
#run model builder test
!python /content/gdrive/"My Drive"/models_teddy/research/object_detection/
 ↪builders/model_builder_tf2_test.py
```

0.3 Step 3: Prepare Image Data

[5]:
```
os.chdir('/content/gdrive/My Drive/')
```

[ ]:
```
print(os.getcwd())
os.chdir('models_teddy')
print(os.getcwd())
```

[ ]:
```
import os
import glob
import pandas as pd
import xml.etree.ElementTree as ET

def xml_to_csv(path):
    xml_list = []
    classes_names = []

    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            print(member[0].text)
            value = (root.find('filename').text,
                     int(root.find('size')[0].text),
                     int(root.find('size')[1].text),
                     member[0].text,
                     int(member[4][0].text),
                     int(member[4][1].text),
                     int(member[4][2].text),
                     int(member[4][3].text)
                     )
            xml_list.append(value)
    column_name = ['filename', 'width', 'height', 'class', 'xmin', 'ymin',␣
 ↪'xmax', 'ymax']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    classes_names = list(set(classes_names))
    classes_names.sort()

    return xml_df, classes_names

def main():
    for directory in ['train', 'test']:
        print(directory)
        image_path = os.path.join(os.getcwd(), 'research/object_detection/images/
 ↪{}'.format(directory))
        xml_df, classes = xml_to_csv(image_path)
        print(os.getcwd())
```

```
        xml_df.to_csv('research/object_detection/data/{}_labels.csv'.
 ↪format(directory), index=None)
        print('Successfully converted xml to csv.')
        print(classes)


main()
```

[ ]:
```
# Convert csv to tf record files - To switch to a code based version so that the␣
 ↪sys.exit dont take place.

from __future__ import division
from __future__ import print_function
from __future__ import absolute_import

import os
import io
import pandas as pd
import tensorflow as tf

from PIL import Image
from object_detection.utils import dataset_util
from collections import namedtuple, OrderedDict

def class_text_to_int(row_label):
    if row_label == 'teddy':
        return 1
    else:
        None

def split(df, group):
    data = namedtuple('data', ['filename', 'object'])
    gb = df.groupby(group)
    return [data(filename, gb.get_group(x)) for filename, x in zip(gb.groups.
 ↪keys(), gb.groups)]

def create_tf_example(group, path):
    print(path)
    print(group.filename)
    with tf.compat.v1.gfile.GFile(os.path.join(path, '{}'.format(group.
 ↪filename)), 'rb') as fid:
        encoded_png = fid.read()
    encoded_png_io = io.BytesIO(encoded_png)
    image = Image.open(encoded_png_io)
    width, height = image.size

    filename = group.filename.encode('utf8')
```

```
    image_format = b'png'
    xmins = []
    xmaxs = []
    ymins = []
    ymaxs = []
    classes_text = []
    classes = []

    for index, row in group.object.iterrows():
        xmins.append(row['xmin'] / width)
        xmaxs.append(row['xmax'] / width)
        ymins.append(row['ymin'] / height)
        ymaxs.append(row['ymax'] / height)
        classes_text.append(row['class'].encode('utf8'))
        classes.append(class_text_to_int(row['class']))
    tf_example = tf.train.Example(features=tf.train.Features(feature={
        'image/height': dataset_util.int64_feature(height),
        'image/width': dataset_util.int64_feature(width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
        'image/encoded': dataset_util.bytes_feature(encoded_png),
        'image/format': dataset_util.bytes_feature(image_format),
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes),
    }))

    return tf_example

def main(_):
    for directory in ['train', 'test']:
      output_path = os.path.join(os.getcwd(), 'research/object_detection/data/{}.
 ↪record'.format(directory))
      print(output_path)
      input_path = os.path.join(os.getcwd(), 'research/object_detection/data/
 ↪{}_labels.csv'.format(directory))
      print(input_path)
      writer = tf.io.TFRecordWriter(output_path)
      print(os.getcwd())
      path = os.path.join(os.getcwd(), 'research/object_detection/images/{}'.
 ↪format(directory))
      print(path)
      examples = pd.read_csv(input_path)
      grouped = split(examples, 'filename')
```

```
        for group in grouped:
            tf_example = create_tf_example(group, path)
            writer.write(tf_example.SerializeToString())

        writer.close()
        print('Successfully created the TFRecords: {}'.format(output_path))

if __name__ == '__main__':
    tf.compat.v1.app.run()
```

[ ]:
```
#checking the pbtxt file
import os
os.chdir('/content/gdrive/My Drive/models_teddy/research/object_detection/data')
print(os.getcwd())
# List all files in the folder
file_list = os.listdir(os.getcwd())
print("List of files in folder:")
for file_name in file_list:
  print(file_name)
!cat object-detection.pbtxt
os.chdir('/content/gdrive/My Drive/models_teddy/research/object_detection')
print(os.getcwd())
labelmap_path = os.path.join(os.getcwd(), 'data/object-detection.pbtxt')
train_record_path = os.path.join(os.getcwd(), 'data/train.record')
test_record_path = os.path.join(os.getcwd(), 'data/test.record')
print(train_record_path)
print(test_record_path)
```

0.4 Step 4: Configuring training

[ ]:
```
batch_size = 16
num_steps = 50000
num_eval_steps = 1000
```

[ ]:
```
!wget http://download.tensorflow.org/models/object_detection/tf2/20200711/
 ↪ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz
!tar -xf ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz
```

[8]:
```
fine_tune_checkpoint = 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8/checkpoint/
 ↪ckpt-0'
```

[ ]:
```
!wget https://raw.githubusercontent.com/tensorflow/models/master/research/
 ↪object_detection/configs/tf2/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.
 ↪config
print(os.getcwd())

base_config_path = 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.config'
```

[ ]:
```
import re

with open(base_config_path) as f:
    config = f.read()

with open('model_config.config', 'w') as f:

  # Set labelmap path
  config = re.sub('label_map_path: ".*?"',
             'label_map_path: "{}"'.format(labelmap_path), config)

  # Set fine_tune_checkpoint path
  config = re.sub('fine_tune_checkpoint: ".*?"',
                  'fine_tune_checkpoint: "{}"'.format(fine_tune_checkpoint),␣
 ↪config)

  # Set train tf-record file path
  config = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED/train)(.*?")',
                  'input_path: "{}"'.format(train_record_path), config)

  # Set test tf-record file path
  config = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED/val)(.*?")',
                  'input_path: "{}"'.format(test_record_path), config)

  # Set number of classes.
  config = re.sub('num_classes: [0-9]+',
                  'num_classes: {}'.format(1), config)

  # Set batch size
  config = re.sub('batch_size: [0-9]+',
                  'batch_size: {}'.format(batch_size), config)

  # Set training steps
  config = re.sub('num_steps: [0-9]+',
                  'num_steps: {}'.format(num_steps), config)

  # Set fine-tune checkpoint type to detection
  config = re.sub('fine_tune_checkpoint_type: "classification"',
             'fine_tune_checkpoint_type: "{}"'.format('detection'), config)

  f.write(config)
```

[ ]:
```
%cat model_config.config
```

[ ]:
```
print(os.getcwd())
!ls
os.chdir('/content/gdrive/My Drive/models_teddy/research/object_detection')
```

```
model_dir = 'training/'
pipeline_config_path = 'model_config.config'
```

0.4.1 Train Model

[ ]:
```
print(os.getcwd())
!ls
```

[ ]:
```
print(pipeline_config_path)
```

```
model_config.config
```

[ ]:
```
!python model_main_tf2.py \
    --pipeline_config_path={pipeline_config_path} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --sample_1_of_n_eval_examples=1 \
    --num_eval_steps={num_eval_steps}
```

[ ]:
```
import os
print(os.getcwd())
%load_ext tensorboard
%tensorboard --logdir 'training/train'
```

0.4.2 Export model inference graph

[ ]:
```
output_directory = 'inference_graph'
print(model_dir)
print(output_directory)
print(pipeline_config_path)
```

[ ]:
```
!python exporter_main_v2.py \
    --trained_checkpoint_dir {model_dir} \
    --output_directory {output_directory} \
    --pipeline_config_path {pipeline_config_path}
```

[ ]:
```
print(model_dir)
```

0.5 Step 5: Test the Model

[ ]:
```
print(os.getcwd())
os.chdir('/content/gdrive/My Drive/models_teddy/research/')
print(os.getcwd())
```

[13]:
```
import io
import os
import scipy.misc
```

```
import numpy as np
import six
import time
import glob
from IPython.display import display

from six import BytesIO

import matplotlib
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw, ImageFont

import tensorflow as tf
from object_detection.utils import ops as utils_ops
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util

%matplotlib inline
```

[ ]:
```
print(os.getcwd())
```

[14]:
```
def load_image_into_numpy_array(path):
  """Load an image from file into a numpy array.

  Puts image into numpy array to feed into tensorflow graph.
  Note that by convention we put it into a numpy array with shape
  (height, width, channels), where channels=3 for RGB.

  Args:
    path: a file path (this can be local or on colossus)

  Returns:
    uint8 numpy array with shape (img_height, img_width, 3)
  """
  img_data = tf.io.gfile.GFile(path, 'rb').read()
  image = Image.open(BytesIO(img_data))
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 1)).astype(np.uint8)
```

[16]:
```
category_index = label_map_util.
↪create_category_index_from_labelmap(labelmap_path, use_display_name=True)
```

[17]:
```
os.chdir('/content/gdrive/My Drive/models_teddy/research/object_detection')
```

[ ]:
```
tf.keras.backend.clear_session()
model = tf.saved_model.load(f'{output_directory}/saved_model')
```

[20]:
```
def run_inference_for_single_image(model, image):
  image = np.asarray(image)
  input_tensor = tf.convert_to_tensor(image)
  input_tensor = input_tensor[tf.newaxis,...]

  # Run inference
  model_fn = model.signatures['serving_default']
  output_dict = model_fn(input_tensor)

  num_detections = int(output_dict.pop('num_detections'))
  output_dict = {key:value[0, :num_detections].numpy()
                 for key,value in output_dict.items()}
  output_dict['num_detections'] = num_detections

  # detection_classes should be ints.
  output_dict['detection_classes'] = output_dict['detection_classes'].astype(np.
↪int64)
  # Handle models with masks:
  if 'detection_masks' in output_dict:
    # Reframe the the bbox mask to the image size.
    detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
              output_dict['detection_masks'], output_dict['detection_boxes'],
               image.shape[0], image.shape[1])
    detection_masks_reframed = tf.cast(detection_masks_reframed > 0.5,
                                       tf.uint8)
    output_dict['detection_masks_reframed'] = detection_masks_reframed.numpy()

  return output_dict
```

[21]:
```
import cv2
```

[ ]:
```
for image_path in glob.glob('images/oof1/*.png'):
  print(image_path)
  image_np = load_image_into_numpy_array(image_path)
  image_np = cv2.cvtColor(image_np, cv2.COLOR_GRAY2RGB)

  output_dict = run_inference_for_single_image(model, image_np)
  vis_util.visualize_boxes_and_labels_on_image_array(
      image_np,
      output_dict['detection_boxes'],
      output_dict['detection_classes'],
      output_dict['detection_scores'],
      category_index,
      instance_masks=output_dict.get('detection_masks_reframed', None),
      use_normalized_coordinates=True,
      line_thickness=8)
  display(Image.fromarray(image_np))
```

Appendix C

Image streamer

```
#!/usr/bin/env python3
import argparse
import socket,struct
import numpy as np
import libs.detection.detection as det

# Args for setting IP/port of AI-deck. Default settings are for when
# AI-deck is in AP mode.
parser = argparse.ArgumentParser(description='Connect to AI-deck JPEG streamer example')
parser.add_argument("-n",  default="192.168.4.1", metavar="ip", help="AI-deck IP")
parser.add_argument("-p", type=int, default='5000', metavar="port", help="AI-deck port")
parser.add_argument('--save', action='store_true', help="Save streamed images")
args = parser.parse_args()

deck_port = args.p
deck_ip = args.n

print("Connecting to socket on {}:{}...".format(deck_ip, deck_port))
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((deck_ip, deck_port))
print("Socket connected")

imgdata = None
data_buffer = bytearray()

def rx_bytes(size):
  data = bytearray()
  while len(data) < size:
    data.extend(client_socket.recv(size-len(data)))
  return data

import cv2

count = 0

model = det.load_model('./inference_graph')
```

```
while(1):
    # First get the info
    packetInfoRaw = rx_bytes(4)
    [length, routing, function] = struct.unpack('<HBB', packetInfoRaw)

    imgHeader = rx_bytes(length - 2)
    [magic, width, height, depth, format, size] = struct.unpack('<BHHBBI', imgHeader)

    if magic == 0xBC:

      # Now we start rx the image, this will be split up in packages of some size
      imgStream = bytearray()

      while len(imgStream) < size:
          packetInfoRaw = rx_bytes(4)
          [length, dst, src] = struct.unpack('<HBB', packetInfoRaw)
          chunk = rx_bytes(length - 2)
          imgStream.extend(chunk)

      count = count + 1

      if format == 0:
          bayer_img = np.frombuffer(imgStream, dtype=np.uint8)
          bayer_img.shape = (244, 324)
          color_img = cv2.cvtColor(bayer_img, cv2.COLOR_BayerBG2BGR)

          cv2.imshow('Raw', bayer_img)

          key = cv2.waitKey(1) & 0xFF

          if args.save or key == ord('s'):
              cv2.imwrite(f"stream_out/test/img_{count:06d}.png", bayer_img)
      else:
          with open("img.jpeg", "wb") as f:
              f.write(imgStream)
          nparr = np.frombuffer(imgStream, np.uint8)
          decoded = cv2.imdecode(nparr,cv2.IMREAD_UNCHANGED)
          cv2.imshow('JPEG', decoded)
          cv2.waitKey(1)
```

Appendix D

Kalman filter

```
import cv2
import numpy as np

# Initialize the Kalman Filter
def initialize_kalman_filter():
    kf = cv2.KalmanFilter(8, 4)
    kf.errorCovPost = np.eye(8, dtype=np.float32) * 1e2

    kf.transitionMatrix = np.array([
        [1, 0, 0, 0, 1, 0, 0, 0],
        [0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 1, 0, 0, 0, 1, 0],
        [0, 0, 0, 1, 0, 0, 0, 1],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1]], np.float32)

    kf.measurementMatrix = np.array([
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0]], np.float32)

    # Define process and measurement noise covariance matrices
    kf.processNoiseCov = np.eye(8, dtype=np.float32) * 5
    kf.measurementNoiseCov = np.eye(4, dtype=np.float32) * 1e-5

    return kf

# Convert the bounding box to a state vector
def bbox_to_state(bbox):
    x, y, w, h = bbox
    return np.array([x+w/2, y+h/2, w, h, 0, 0, 0, 0], dtype=np.float32)

def bbox_to_state_oof(bbox):
```

```
    x, y, w, h = bbox
    return np.array([x+w/2, y+h/2, w, h], dtype=np.float32)

# Convert the state vector to a bounding box
def state_to_bbox(state):
    x, y, w, h, _, _, _, _ = state
    return (int(x-w/2), int(y-h/2), int(w), int(h))
```

Appendix E

Image processing

```python
import time
import cv2
import sys

root_dir = "/Users/artemredchych/bakalar/project-vision"
sys.path.append(root_dir)

import libs.tracking.kalman as kalman
import libs.detection.final˙detection as det
def distance_to_object(
        bbox_height_in_pixels,
        real_object_height=88,
        effective_focal_length=0.66,
        pixel_size=3.6):
    """
    Calculate the distance to the object from the camera.

    :param bbox_height_in_pixels: Height of the object in the image in pixels.
    :param real_object_height: Real height of the object in centimeters.
           Default is 88 cm.
    :param effective_focal_length: Effective focal length of the camera in millimeters.
           Default is 0.66 mm.
    :param pixel_size: Size of a pixel in micrometers. Default is 3.6 µm.

    :return: Distance to the object from the camera in centimeters.
    """
    # Convert pixel size to millimeters
    pixel_size_mm = pixel_size / 1000.0

    # Convert real_object_height to millimeters
    real_object_height_mm = real_object_height * 10

    # Calculate distance to object
    distance = (
        real_object_height_mm * effective_focal_length
    ) / (bbox_height_in_pixels * pixel_size_mm)
```

```
    # Convert distance to centimeters
    distance_cm = distance / 10

    return distance_cm

def init_kalman():
    return kalman.initialize_kalman_filter()

def print_distance(image, bbox, text_color=(0, 0, 0)):
    image_copy = image.copy()
    height, width, _ = image.shape
    ymin, xmin, ymax, xmax = bbox
    ymin = int(ymin * height)
    ymax = int(ymax * height)
    xmin = int(xmin * width)
    xmax = int(xmax * width)

    distance = distance_to_object(ymax - ymin)

    dist_label_x = int(xmin + 5)
    dist_label_y = int(ymin + ((ymax - ymin) / 2))
    dist_label = f'dist: {distance:.2f} cm'
    cv2.putText(
        image_copy,
        dist_label,
        (dist_label_x,
         dist_label_y),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.5, (0, 153, 255), 1
    )

    return image_copy

def process_image(model, image, kf, is_first):
    # Run inference on the image
    start_time = time.perf_counter()
    output_dict = det.run_inference_for_single_image(model, image)
    end_time = time.perf_counter()
    execution_time = end_time - start_time
    execution_time_ms = execution_time * 1000  # Convert seconds to milliseconds
    print(f'Function execution time: {execution_time_ms:.2f} milliseconds')

    # Draw bounding boxes and class names on the image
    detection = det.get_bounding_box(output_dict=output_dict, image=image, threshold=0.9)
    if detection is not None:
        bbox, class_name, score = detection
        print(f"Detected bounding box: {bbox}")
        print(f"score: {score}")
        print(f"class_name: {class_name}")
        result_image = det.draw_detection_bbox(image, bbox, score, class_name)
        if is_first:
```

```
                initial_bbox = bbox
                # Set the initial state
                kf.statePost = kalman.bbox_to_state(initial_bbox)
            else:
                predicted_state = kf.predict()
                predicted_box = kalman.state_to_bbox(predicted_state)
                print(f"Predicted bounding box: {predicted_box}")

                # Update the state with the new measurement
                kf.correct(kalman.bbox_to_state_oof(bbox))

                # Get the current state and convert it to a bounding box
                correct_state = kf.statePost
                correct_bbox = kalman.state_to_bbox(correct_state)
                print(f"After correction: {correct_bbox}\n")

                result_image = det.draw_predicted_bbox(result_image, predicted_box, (0, 0, 255))
                result_image = det.draw_predicted_bbox(result_image, correct_bbox, (255, 0, 0))
                print(det.calculate_iou(bbox, kalman.state_to_bbox(predicted_state)))

            return result_image

        return None
```

Appendix F

Extended possition commander

```
import time
import math
from cflib.positioning.position_hl_commander import PositionHlCommander

class PositionExtendedCommander(PositionHlCommander):
    # seconds for 1 degree
    TURNING_SPEED = 0.03
    DEFAULT = None

    def __init__(self, crazyflie,
                 x=0.0, y=0.0, z=0.0, yaw=0,
                 default_velocity=0.5,
                 default_height=0.5,
                 controller=None,
                 default_landing_height=0.0):
        super().__init__(crazyflie,
                         x=x, y=y, z=z,
                         default_velocity=default_velocity,
                         default_height=default_height,
                         controller=controller,
                         default_landing_height=default_landing_height)
        self._yaw = yaw

    def __rotate(self, yaw_angle_degrees):
        x = self._x
        y = self._y
        z = self._z
        yaw = self._yaw + math.radians(yaw_angle_degrees)

        duration_s = math.fabs(yaw_angle_degrees * self.TURNING_SPEED)
        self._hl_commander.go_to(x, y, z, yaw, duration_s, False)
        time.sleep(duration_s)
        self._yaw = yaw

    def turn_left(self, angle):
        self.__rotate(math.fabs(angle))
```

```
def turn_right(self, angle):
    self.__rotate(-1 * angle)

def left(self, distance_m, velocity=DEFAULT):
    """
    Go left

    :param distance_m: The distance to travel (meters)
    :param velocity: The velocity of the motion (meters/second)
    :return:
    """
    perpendicular_angle = self._yaw + math.pi / 2
    x_component = math.cos(perpendicular_angle) * distance_m
    y_component = math.sin(perpendicular_angle) * distance_m
    self.move_distance(x_component, y_component, 0.0, velocity)

def right(self, distance_m, velocity=DEFAULT):
    """
    Go right

    :param distance_m: The distance to travel (meters)
    :param velocity: The velocity of the motion (meters/second)
    :return:
    """
    perpendicular_angle = self._yaw + math.pi / 2
    x_component = math.cos(perpendicular_angle) * distance_m
    y_component = math.sin(perpendicular_angle) * distance_m
    self.move_distance(-x_component, -y_component, 0.0, velocity)

def forward(self, distance_m, velocity=DEFAULT):
    """
    Go forward

    :param distance_m: The distance to travel (meters)
    :param velocity: The velocity of the motion (meters/second)
    :return:
    """
    dist_x = distance_m * math.cos(self._yaw)
    dist_y = distance_m * math.sin(self._yaw)
    self.move_distance(dist_x, dist_y, 0.0, velocity)

def back(self, distance_m, velocity=DEFAULT):
    """
    Go backwards

    :param distance_m: The distance to travel (meters)
    :param velocity: The velocity of the motion (meters/second)
    :return:
    """
    dist_x = distance_m * math.cos(self._yaw)
    dist_y = distance_m * math.sin(self._yaw)
```

```
        self.move_distance(-dist_x, -dist_y, 0.0, velocity)

    def up(self, distance_m, velocity=DEFAULT):
        """
        Go up

        :param distance_m: The distance to travel (meters)
        :param velocity: The velocity of the motion (meters/second)
        :return:
        """
        self.move_distance(0.0, 0.0, distance_m, velocity)

    def down(self, distance_m, velocity=DEFAULT):
        """
        Go down

        :param distance_m: The distance to travel (meters)
        :param velocity: The velocity of the motion (meters/second)
        :return:
        """
        self.move_distance(0.0, 0.0, -distance_m, velocity)

    def move_distance(self, distance_x_m, distance_y_m, distance_z_m,
                      velocity=DEFAULT):
        """
        Move in a straight line.
        positive X is forward
        positive Y is left
        positive Z is up

        :param distance_x_m: The distance to travel along the X-axis (meters)
        :param distance_y_m: The distance to travel along the Y-axis (meters)
        :param distance_z_m: The distance to travel along the Z-axis (meters)
        :param velocity: The velocity of the motion (meters/second)
        :return:
        """

        x = self._x + distance_x_m
        y = self._y + distance_y_m
        z = self._z + distance_z_m

        self.go_to(x, y, z, velocity)

    def go_to(self, x, y, z=DEFAULT, velocity=DEFAULT):
        """
        Go to a position

        :param x: X coordinate
        :param y: Y coordinate
        :param z: Z coordinate
        :param velocity: The velocity (meters/second)
        :return:
```

```
    """

    z = self._height(z)

    dx = x - self._x
    dy = y - self._y
    dz = z - self._z
    distance = math.sqrt(dx * dx + dy * dy + dz * dz)

    if distance > 0.0:
        duration_s = distance / self._velocity(velocity)
        self._hl_commander.go_to(x, y, z, self._yaw, duration_s)
        time.sleep(duration_s)

        self._x = x
        self._y = y
        self._z = z
```

Appendix G

Object detection and tracking streamer

```
#!/usr/bin/env python3
import argparse
import socket, struct
import numpy as np
import libs.detection.detection as det
import libs.image_processing as ip
import libs.tracking.kalman as kalman
import time
import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from libs.pos.pos_commander import PositionExtendedCommander
from cflib.utils import uri_helper

# Args for setting IP/port of AI-deck. Default settings are for when
# AI-deck is in AP mode.
parser = argparse.ArgumentParser(description='Connect to AI-deck JPEG streamer example')
parser.add_argument("-n",  default="192.168.4.1", metavar="ip", help="AI-deck IP")
parser.add_argument("-p", type=int, default='5000', metavar="port", help="AI-deck port")
parser.add_argument('--save', action='store_true', help="Save streamed images")
args = parser.parse_args()

deck_port = args.p
deck_ip = args.n

print("Connecting to socket on {}:{}...".format(deck_ip, deck_port))
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((deck_ip, deck_port))
print("Socket connected")

imgdata = None
data_buffer = bytearray()

def rx_bytes(size):
```

```
    data = bytearray()
    while len(data) < size:
        data.extend(client_socket.recv(size-len(data)))
    return data

import cv2

start = time.time()
count = 0
exit_all = False

uri = uri_helper.uri_from_env(default='radio://0/80/2M/E7E7E7E7E5')
model = det.load_model('./inference_graph')
is_first = True
kf = kalman.initialize_kalman_filter()
cflib.crtp.init_drivers()

with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
    last_x = 0.8
    last_y = 0.9
    last_z = -0.3
    with PositionExtendedCommander(
            scf,
            x=last_x, y=last_y, z=last_z, yaw=0,
            default_velocity=0.1,
            default_height=last_z + 0.5,
            controller=PositionExtendedCommander.CONTROLLER_PID,
            default_landing_height=last_z
    ) as pc:
        #go to init position
        pc.up(0.6)
        pc.right(0.9)

        frames_with_bear = 0
        frames_without_bear = 0
        #start object detection and tracking
        while(1):
            # First get the info
            packetInfoRaw = rx_bytes(4)
            [length, routing, function] = struct.unpack('<HBB', packetInfoRaw)

            imgHeader = rx_bytes(length - 2)
            [magic, width, height, depth, format, size] = struct.unpack('<BHHBBI', imgHeader)

            if magic == 0xBC:
                imgStream = bytearray()

                while len(imgStream) < size:
                    packetInfoRaw = rx_bytes(4)
                    [length, dst, src] = struct.unpack('<HBB', packetInfoRaw)
                    chunk = rx_bytes(length - 2)
                    imgStream.extend(chunk)
```

```
bayer_img = np.frombuffer(imgStream, dtype=np.uint8)
bayer_img.shape = (244, 324)
color_img = cv2.cvtColor(bayer_img, cv2.COLOR_BayerBG2BGR)
result_image = ip.process_image(
    model=model,
    image=color_img,
    kf=kf,
    is_first=is_first
)

if result_image is not None:
    is_first = False
    frames_with_bear += 1
    cv2.imshow('result', result_image)
else:
    frames_without_bear += 1
    if frames_without_bear > 10:
        pc.turn_left(15)
        frames_without_bear = 0

    cv2.imshow('result', color_img)

count += 1
key = cv2.waitKey(1) & 0xFF
if key == ord('s'):
    break
```

Contents of the attached medium

- README.md
- text
  - thesis.pdf ..... text of this thesis in PDF format
  - thesis ..... source of this thesis LaTeX
- data
  - lab_test_raw ..... raw data from the tests
  - lab_test_result ..... detection and tracking results
  - training ..... training dataset with images and labels
- inference_graph ..... stored model for object detection
- crazyflie-lib-python ..... python library for crazyflie
- libs
  - detection ..... scripts used in object detection
  - pos ..... scripts used in drone flying
  - tracking ..... scripts used in object tracking
  - models ..... tensor flow libraries
- utility ..... useful scripts
- flying_streamer.py ..... real-time object detection and tracking streamer
- requirements.txt ..... python requirements
- result.mp4 ..... detection and tracking result in MP4 format
- result.mp4 ..... detection and tracking result in GIF format
- lab_test_video.MOV ..... video of the flying drone from the laboratory