Individual Thesis

DEVELOPING A LIGHTWEIGHT COUNTER-DRONE SYSTEM USING COMPUTER VISION AND ROBOT OPERATIONG SYSTEM

Kyriakos Poyiatzi

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Developing a lightweight counter-drone system using computer vision and robot operating system

Kyriakos Poyiatzi

Supervising Professor:
Dr Panagiotis Kolios

The Individual Thesis was submitted in partial fulfillment of the requirements for the degree of Computer Science of the Department of Computer Science of the University of Cyprus.

May 2025

Acknowledgments

I would like to first of all give special thanks to my supervisor of my Individual Thesis, Dr Panagiotis Kolios, for the guidance and support he gave me throughout the whole thesis, helping me to fully understand the objectives and ideas of the system.

In addition, I would like to thank the members of the Research Center "Koios", Nikolas Soulis and Yannis Grigoriou, who although they had to face their personal projects on a daily basis, helped me tremendously to understand the current system and to acclimatize myself so that I could complete the system. Furthermore, I would like to thank my fellow student and member of Research Center "Koios" George Pettemerides for his support and his help throughout the experiments and tests of my thesis. I would also like to thank all of them for the excellent cooperation we had during that period.

Finally, I would like to thank my family, friends and fellow students who pushed me every day, gave me energy and courage in the difficult times to achieve my goals. Again, I thank each and every one of them individually as without these individuals I mentioned above, implementing the system would have been much more difficult for me.

Abstract

This thesis project presents modifications and enhancements to a partially implemented drone detection and tracking system originally developed by an ex-employee of “Koios” centre of excellence, Rafael Makrigiorgis. The system is based on a trained YOLOv4 model to detect and track drones in real time, identify their types, and estimate their distance and speed. It has also multi-detection capabilities, enabling the tracking of multiple drones simultaneously, while dynamically computing an aggregate detection region to maintain targets within the camera’s FOV. Two distinct deployment environments are considered.

The first implementation runs on an NVIDIA Jetson Xavier connected to a DJI-Matrice-300-RTK with a DJI-Zenmuse-H20T camera, where live video from the drone’s camera is analysed using the Robot Operating System (ROS) and OpenCV. In this configuration, the system also integrates an active visual feature. When a target gets out of the FOV (Field of View), a PID-based control loop computes yaw commands that are sent to the drone, autonomously re-cantering the target.

The second implementation, designed for a standard CPU/GPU setup (like any PC or Laptop), processes pre-recorded videos for drone detection and tracking using similar techniques. It employs a SORT tracker for multi-object tracking and applies the pinhole camera model and kinematic equations to calculate distance and speed. The enhancements not only improve detection accuracy and tracking robustness across different operational platforms but also offer the potential for fully autonomous drone manoeuvring based on real-time visual feedback.

The systems were developed using Python (2.7 for the jetson version and 3.13 for the pc version), with key components trained on YOLOv4 and YOLOv4-tiny via Darknet and Google Collab and tested using simulated scenarios to ensure both safety and performance.

Table of Contents

Chapter 1

Introduction

1.1 Background Information and Motivation

Drone technology that has the potential to disrupt and augment our quality of life is swiftly evolving. A drone, formally also known as Unmanned Aerial Vehicles (UAVs), are plane-robots that do not have any passengers or pilots on board. Such "flying objects" are controlled either by man with the use of physical controller or fly autonomously according to pre-written code in its program from its onboard sensors and GPS. Considering the short history of UAVs, UAVs were closest to the military to use as spy planes and precision strikes but now the drones have developed very rapidly, and drones are now making humanity's life easier in plenty of new areas such as emergency services, agriculture and farming, science exploration, aerial photography, etc.

Their rapid growth also unveiled enormous security risks and privacy risks. As remote-operating and autonomous vehicles, they are vulnerable to malicious use for the purposes of espionage, contraband smuggling, and even terrorism. Drones were used to smuggle contraband across the border illicitly, interfere with critical infrastructure, and even launch targeted strikes. Their use of controlled airspace is even posing serious safety risks to aviation security with numerous cases of near-misses with commercial flights reported. Even the risk of cyber-attack is on the cards with hostile users gaining the ability to hijack drones to carry out malicious actions. One example is an incident that took place in London's Gatwick Airport back in 2018 when two drones were hovering over the airport. This caused the cancellation of more than fifty flights and the fear of more than 100,000 passengers. Existing counter-drone systems are often expensive, complex, and unsuitable for small-scale or embedded use, so there is a clear need for affordable, lightweight, and real-time detection systems based solely on visual input.

These new risks and threats cause the urgent need for advanced drone detection systems that will effectively detect, track, identify and neutralize unwanted UAVs over vulnerable areas.

These kinds of systems need to be implemented to protect the safety of the citizens, national security, and privacy in the current era of drones becoming cheaper and advanced.

Over my four years at the University of Cyprus, I've always been fascinated by how technology can be used to solve real-world problems. I didn't want my thesis to be just research on a random topic just to be graded. I wanted to implement a system that would have an impact on people's lives. With more prominence given to the use of drones, I explored their vulnerabilities as a concern and observed the ways in which excessive use of drones has escalated as a cause of concern.

My goal is to create a new system which will be similar to the existing system. The new system will have new critical functions and features in order to optimize the performance and accuracy of the detections. It is also necessary to implement the system in such a way so that experts in the field can further develop and improve the system in the future.

1.2 Problem Statement

Despite the breakthrough in drone detection technologies, challenges are numerous. Drones are small targets that operate at low altitude at high speeds and therefore it is quite difficult to track them by the usage of just traditional radars. As already mentioned in the previous section, drones come in relatively small sizes (There are also larger drones used mostly in the army) which makes also it difficult to detect them and separate them from birds or other airborne objects. This is also essential for such systems and nowadays with the usage of advanced Machine Learning models that support real-time classification we can achieve it.

Interference and spoofing of the signals when the aggressors attempt to hide their drones or destroy detection equipment is also the obstruction. Neutralization of malicious drones is both technology- and legally challenging as unlawful anti-measures (GPS jamming) interfere with legitimate aerial activities. These require breakthroughs in lightweight and scalable detection hardware that is capable of providing accurate real-time localizations that will not interfere with the safety of the flight of drones.

AI-based detection systems that use computer vision, machine learning, and multi-sensor fusion are in the making to improve the accuracy and response times, because these two play the central roles in areas like airports, government premises, open gatherings, and defense campuses where discovery of damages is of utmost importance. Such advanced systems also have some challenges as their accuracy and performance is crucial and needs to be almost perfect in order for them to be useful.

In general, though the numerous uses of drones in numerous sectors are highly advantageous, their malicious usage is of utmost concern to security. Ever-growing risks of illegal drone usage characterize the pressing need for the utilization of high-tech detection systems.

1.3 Research Objectives

The main objective of the thesis is to improve the existing system which is at a very early stage. However, despite the improved version which is compatible for an NVIDIA Xavier Jetson, there is another version which is compatible for Windows/Linux/MacOS environments. A drone detection and tracking system should be developed, which will have the ability to classify, identify, track and detect multiple drones in real-time by integrating computer vision, machine learning, and multi-sensor fusion. For the implementation of the system, some key technologies were used like OpenCV, ROS (Robot Operating System), Numerical python, Twist for drone movement control, YOLOv4, SORT and Deque for tracking. The system has the following features

1. **Real-Time Drone Detection:** The main goal of the system is to detect and track UAVs in real-time via live streaming or videos. For this feature we will rely on the training of a YOLOv4 model for drone detection, in which we will classify the drones by types. Also, we will implement OpenCV's Deep Neural Network in order to load and run the YOLOv4 model effectively.
2. **Multi-Detection of Drones and Tracking:** Multi-Drones detection is essential for such systems, as in most cases more than one drone is used for a malicious activity. To achieve this feature in both versions we will need to use two different object tracking algorithms. The first one for the laptop environment is SORT (Simple Online and Realtime Tracker) and Deque (Double-Ended Queue) for the Jetson environment. Both of these algorithms allow efficient tracking of drone's motion.
3. **Drone's Information:** In order for a system to be useful we also need to provide some additional information for each drone like the drone's distance from the detector's camera which will be calculated using the Pinhole Camera Model Equation, its speed which is also calculated using a mathematical formula, the Basic Kinematic Equation. Additionally, because of the training approach that we choose, we will also be able to identify each drone's type. This will also help us to learn the size of the drone, something that we need for the distance and speed.
4. **PID-Based Drone Flight Control:** Modern drones are fast moving flying objects that can do maneuvers, hide under or behind other objects and in general they have a lot of capabilities. Because of that, it is almost impossible to have a human being behind a controller at all times to try and follow the direction of the detected drone. The main device that this system will be installed on is actually a drone controlled by the us. By implementing a PID controller which will adjust the drone's positioning to follow the detected object will be ideal and far more effective than having someone control the

drone's positioning manually. We will use some ROS services in order to achieve dynamical movement like JoystickAction and ObtainControlAuthority.

1.4 Thesis Structure

This thesis consists of six chapters:

1. **Chapter 1 – Introduction:** This chapter provides background information about drone technologies, the motivation for this study, the key problem statement, and the overall study objectives.
2. **Chapter 2 – Literature Review:** In this chapter we will take a look on the history of drone technologies and threats, machine learning and traditional detection methods, and current associated systems both in military and civilian contexts.
3. **Chapter 3 – Previous System:** Chapter 3 outlines the design and implementation of the previous drone detection and tracking system designed by old members of the center of excellence "Koios", including shortcomings and how it can be improved.
4. **Chapter 4 - Implementation and Methodology:** This is the main part of the dissertation, which describes the design of the new detection system designed for the purposes of the thesis, its structure, the main components, the algorithms, and the design choices that were made in order to maximize functionality and performance.
5. **Chapter 5 – Evaluation and Results:** We will have a look at the testing procedures, the evaluation metrics of the training and in general the results of the experiments that were made.
6. **Chapter 6 – Conclusion and Future Work:** The final chapter of the thesis is all about conclusions about the system that was implemented, future work that will be useful and in general some setbacks we faced during the development of the system and its current limitations

Chapter 2

Literature Review

This chapter provides a comprehensive overview of existing research, theories and technological developments related to the drone detection and tracking system. Some more important aspects will be presented in depth such as what a drone can do and how it becomes dangerous, the importance of drone detection, traditional detection methods and how artificial intelligence has changed the concept. Finally, some modern, state-of-the-art detection systems created by some of the world's largest companies will be mentioned.

2.1 Overview of Drone Technologies and Applications

The main reason for implementing this system is the concerns that scientists and experts are beginning to have about what a drone is actually capable of doing. The concept of a drone and the areas where a drone is useful in our daily life will be analyzed, but also the dangers that can be caused by it.

2.1.1 Evolution of UAVs

Unmanned Aerial Vehicles as explained in the beginning of the thesis are aircraft without on-board crew or passengers. UAVs were initially designed for military purposes back in WWI for targeted attacks and recognition. The first modern drone was the RQ-1 Predator which was created by the United States in the 1990s providing real-time surveillance and precision strikes. In the early 2000s the drone industry expanded beyond the military as drones became available for commercial use. Their rapid expansion worldwide occurred around the 2010s with giants such as DJI, Parrot and 3D Robotics dominating the market with the manufacture of increasingly modern drones, which can meet important human needs with great ease and accuracy. [1]

2.1.2 Civilian and Commercial Applications of UAVs

Drones in the recent years, have had a significant role in the development of numerous industries, as the use of them increased productivity, efficiency and cost reduction. The most important industries impacted are:

- **Agriculture and precision farming:** Drone technology has completely revolutionized agriculture, turning it into "precision agriculture". Equipped with multi-spectral and thermal sensors, UAVs allow farmers to monitor the health of their crop, control irrigation and detect parasites or diseases before they spread in order to protect their products from disaster. Spray drones reduce the waste of pesticides and fertilizers, which makes farming more efficient and sustainable. The use of artificial intelligence and machine learning also improves decision-making in farm management, which leads to better yields and reduced costs.
- **Disaster Response & Emergency Services:** One of the most important areas affected in a good way by drones is undoubtedly the Emergency Services. Such services are firefighting, search and rescue and natural disasters. For example, in firefighting they play a crucial role as they provide extremely useful features like thermal imaging which help detect hotspots and guide firefighters to critical areas.

Some additional examples of industries that are using drones frequently are: Construction & Infrastructure, Logistics & Delivery Services, Security & Surveillance, Media & Entertainment, Environmental Conservation, Energy & Utilities, Scientific Research & Space Exploration. [2]

2.1.3 Security Risks Caused by UAVs

Drones provide countless benefits. Unfortunately, a lot of people misuse and take advantage of the advanced capabilities of drones. This is causing concern among experts about the way this situation can be controlled as it’s getting out of hand. Some of these security challenges are [3]:

- **Espionage and Unauthorized Surveillance**

  Most drones are equipped with multiple advanced sensors and high-quality cameras that can be used for unauthorized surveillance in prohibited or sensitives areas like military bases, government buildings and private properties. Those drones can steal data through malicious hardware without anyone noticing.
- **Smuggling and Contraband Transport:**

  Cartels and criminal organizations are using drones to transport their illegal products across the borders.

- **Aviation Threats and Airspace Violations:**

  One of the most serious threats has to do with aviation as drones fly into unauthorized spaces putting at risk human lives. There have been many incidents in the last decade like the Gatwick Airport incident in 2018, the Dubai Airport disruption in 2016 and the near-miss at the Heathrow Airport in 2016 again.

- **Cybersecurity Threats and Drone Hijacking:**

  Another critical challenge not directly related to the drone user is when a hacker takes control of these drones. This is because drones rely on GPS and wireless communication, they are vulnerable to cyber-attacks.

2.2 Drone Detection and Counter Drone Technologies

As mentioned in section 2.1.3, UAVs offer not only benefits but also, if used incorrectly, many safety issues and threats. The current section provide an in-depth examination on how to control and avoid these issues by using drone detection systems. Also, it digs into the importance of drone detection, traditional detection methods, and advancements in AI/ML-based detection technologies.

2.2.1 Importance of Drone Detection

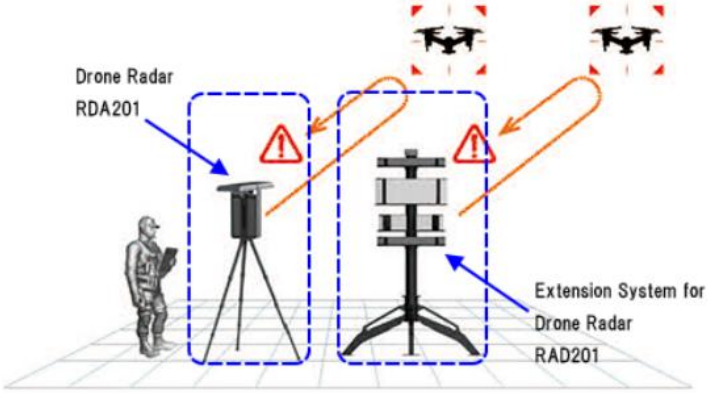
Why is “Drone Detection” so important and what connection does it have with the security risks of drones? The answer is simple. Imagine if there was a system that would alert you directly when a UAV entered an unauthorized area or, even better, prevented it from entering the area. Such systems will protect sensitive areas and ensure safety for the public. Key reasons for drone detection include [4]:

- **Protection of Critical Infrastructure:** In most cases, the areas being attacked by UAVs have important facilities such as data centers, government buildings, banks, military zones. These areas are of high importance and must be protected at all times to ensure data integrity. Effective drone detection systems can prevent unauthorized access and potentially secure the facilities.
- **Airspace Safety:** A major concern is the presence of unauthorized UAVs in airspace (airports, military bases) as it can collide with commercial airlines which carry lives on-board. A drone detection system is crucial for such places as it will ensure that flights are safe by alerting the airports authorities for possible threats. In addition to that, more advanced drone detection systems will be able to prevent the UAVs from entering the airspace with the use of countermeasures like jamming.[5]
- **Public Event Security:** Drone detection systems are a must also for public events in which thousands of people join. Such events are targets for terrorist's attacks, which can also happen very easily with drones. The system on these occasions should be able to detect any kind of UAV that is not authorized from a respective distance in order to inform the authorities on time for further actions.
- **Cybersecurity Concerns:** One of the most common and relatively easy ways of hacking wireless networks is by using a drone. The only thing the middleman has to do (middleman is the hacker) is to install the hacking tools and algorithms on the drone or by using external hardware on it. Without a drone detection system to inform that a suspicious drone is trespassing the private area no one will really have a clue of what happened until it is too late.

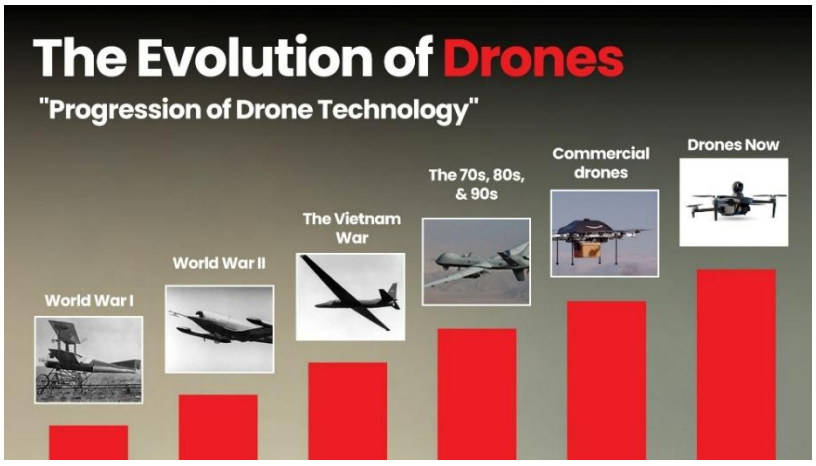
2.2.2 Traditional Drone Detection Methods

Before the rapid development of technology that brought us machine learning and artificial intelligence, IT professionals created some traditional drone detection methods (mostly radar systems). Some years ago, these methods would have been particularly useful to meet the needs we mentioned before, however drones today evolved a lot since the implementation of those methods and because of that they are not particularly useful as they have a lot of limitations. To be more specific let's introduce and explain these traditional methods [8]:

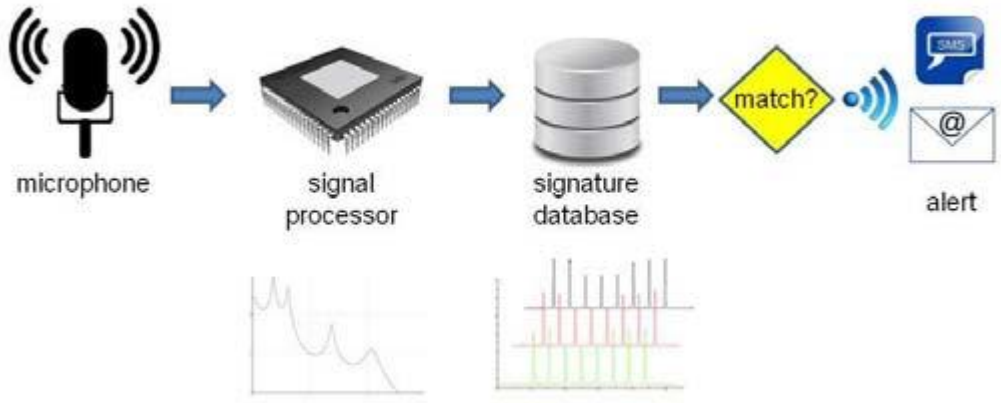
- **Radar Systems:** These types of systems are based on signals. A radar system emits radio waves within a certain range. When those radio waves reach an object, then the object reflects the signal back to the sender. This allows the radar system to locate and detect the unwanted objects that appear near the area. In addition, these systems are capable of detecting objects, measure their speed and direction at long distances without getting affected by any weather. [6][8]



➔ **Limitations of Radar Systems:** Despite their advantages, radar systems are quite expensive, and their deployment is very complex. Another limitation of these systems is that it would be very difficult to detect a small drone that flies at a low altitude as small drones have low radar cross-section. We can see in the image below how drone's sizes have evolved throughout the years. In most cases the drones will not exceed 1 meter, which makes the radar systems useless. [7, 8]



- **Acoustic Sensors:** Acoustic sensors are much different from other traditional detection methods as they don't use signals or radio waves to detect drones. They depend solely on the capturing of a drone's unique sounds, like its motor or the propellers. Let's explain shortly how these sensors operate. Acoustic systems use arrays of microphones to monitor ambient sounds and consist of a database of known drone sounds, facilitating identification and tracking. When a drone comes close to the sensors, the sound generated by the drone is collected in order to compare it with the database. These systems are particularly helpful when the drone doesn't emit radio signals or when the RF detection is challenging in the specific environment. [9]

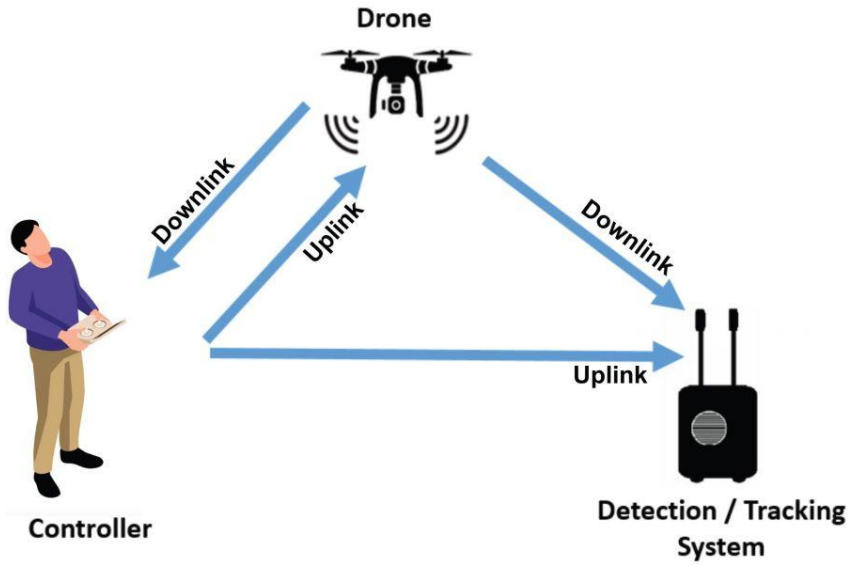


➔ **Limitations of Acoustic Sensors:** Like all the traditional detection systems, acoustic sensors also have their limitations. First of all, the sounds can be collected for drones that are at most 500 meters away from the sensors. For example, airports are much larger than 500 meters, which means such sensors won't work unless there is more than 1. Additionally, noise affects the system's efficiency as there might be false positives or false negatives in noisy and extremely silent environments (false positives and false negatives are basically when the detection is wrong). [9]

- **Radio Frequency (RF) Analysis:** Radio frequency analysis is a widespread, cost-effective method for detecting drones by monitoring their communication signals. These communication signals are transmitted between the drone and its remote controller. In simple words, RF analysis works by intercepting those signals, which is a key factor to detect the drone and its operator (the fact that we can also track the operator of the drone is a huge advantage). Other than that, via RF analysis we have the ability to collect information about drones, like type, model and specs.[10]

➔ **Limitations of RF Analysis:** This method is ineffective when the drone is autonomous, like in most drone terrorist attacks, as there are not communications

signals to retrieve. In addition, like the detection system with acoustic sensors, it has a limited range in which it can intercept signals and is affected by high radio traffic, like in airports.[10]



- **Optical Systems:** Optical systems, as indicated by their name, utilize visual technologies to detect and track drones with the help of cameras. This method plays a pivotal role in the development of drone detection systems as it's the first time that there is no need for signal analysis. This method works like a camera system, in which we can see the drones getting detected in real-time. The advantages of this method in comparison to the others are that we have for the first time visual confirmation that a drone is actually present in the area. Subsequent chapters will demonstrate that optical systems can become very effective when we combine ML and AI to it, which is something applies on the system of this thesis [14].
  - ➔ **Limitations of Optical System:** Despite the fact that these types of systems are the best fit for modern drone detection, there are some limitations we need to address. Weather conditions like fog, rain and low light affect the detections of the system. Also, it requires a clear line of sight in order to be able to detect and identify the target. In addition to that, the reason we need AI and ML is because with a simple optical system there are a lot of false alarms as there is confusion with birds and other objects that look like drones [15].

2.3 AI-Based Drone Detection

AI-Based Drone Detection takes advantage of sophisticated algorithms in order to track and detect UAVs. By building detection systems using Artificial Intelligence, they can become more precise and reactive compared to traditional methods like we studied in chapter 2.2.2.

This section explores how machine learning improves drone detection by utilizing visual, acoustic, and radio frequency data.

2.3.1 Machine Learning

Machine Learning is a crucial factor for AI-Based drone detection systems, as it enables more accurate and efficient systems by training them in order to identify the desired targets. In order to create a Machine Learning model, there is the need for data collection, like images, sounds or even signal patterns (according to the detection method chosen). After the creation of the dataset, the system should be trained to become capable of recognizing such data or similar data in real time [14].

Three categories of machine learning-based detection systems will be discussed:

- ➔ **Visual Detection:** For this type of detection, the algorithm is processing images captured by cameras in order to detect UAVs based on their characteristics. This kind of detection will be applied to the thesis system as it's the most effective way to detect drones in our case. For this thesis, there is the need to collect a dataset of drone images, and the aim is to train the system with that dataset in order to identify and detect drones with a camera stream.[11]
- ➔ **Acoustic Detection:** Such systems were analyzed already in section 2.2.2. The addition of machine learning in acoustic detection is a game-changing factor as with the correct dataset and training the system will be able to analyze the sound patterns and identify every drone based on its sound with limited errors.[12]
- ➔ **RF Analysis:** RF analysis can also be improved with the integration of machine learning as a huge dataset of every possible signal the drones emit with their controllers is needed. These kinds of systems will be able to classify the different types of drones by learning their signal's characteristics.[13]

The main reason for selecting the implementation of visual detection in this thesis project is based on the availability of data and the ease of collecting them and also the fact that visual models can be expanded at all times with new training data if needed and because it can work in various environments without the need of signal/sound analysis like the other two types of systems.

In general, we can characterize these systems as "adaptable" because ML models can learn from complex datasets, allowing them to adapt in any similar scenario without any difficulty. To achieve that, the content of the dataset must be balanced, precise and large as we need the system to be as close to perfection as possible.

Machine learning, while highly useful, doesn't come without its fair share of challenges. One of the most important issues has to do with the quality of the dataset. If the training data is not

accurate, well-balanced, or clean, then the model will not perform appropriately in real situations. Having enough data is also necessary for the model to learn successfully, otherwise it may miss important patterns or overfit. Another issue is that some of these models, especially in deep learning, are computationally expensive. This becomes an issue when we need rapid, real-time feedback, e.g., for drone detection. Finally, models can become disoriented in confusing scenarios, e.g., strange areas or things that look like drones. These are important points to consider when designing any AI-based detection system.

2.4 Object Detection Techniques for UAVs

Object Detection Techniques for UAVs section explores the various techniques used for drone detection in real-time using computer vision and artificial intelligence. To be more exact, we will take a closer look at Deep Learning, in Convolutional Neural Networks, in YOLO and in Feature-Based detection.

2.4.1 Deep Learning for Object Detection

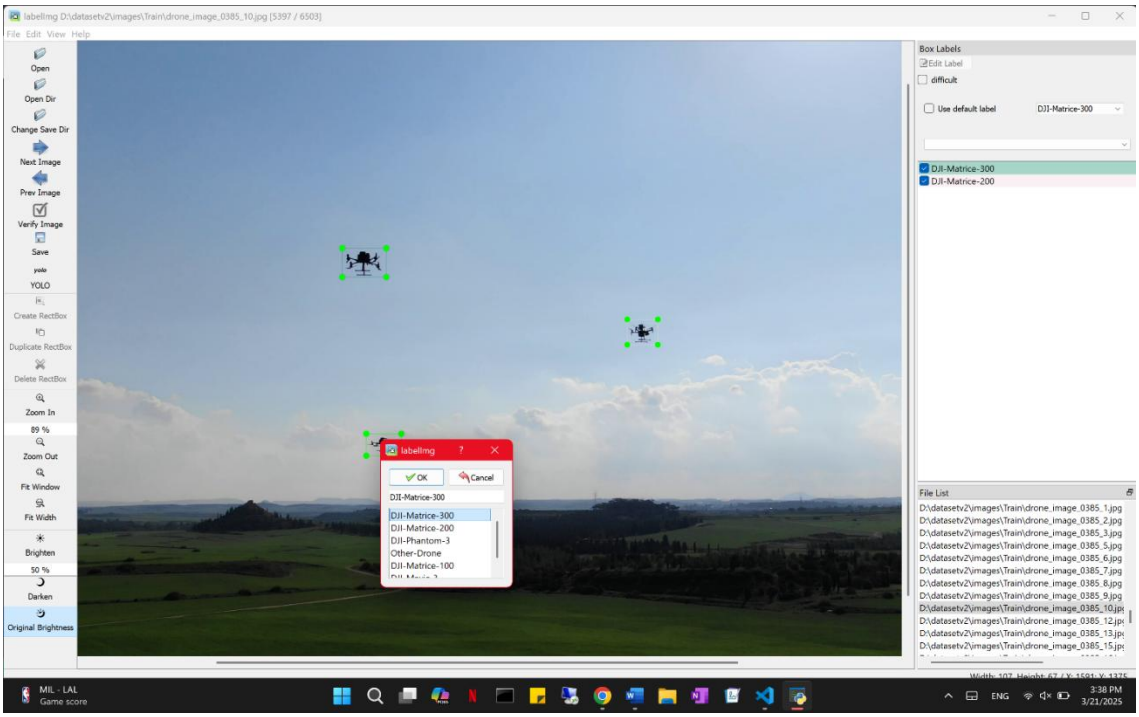
Deep Learning has revolutionized the art of object detection in general, but specifically for dynamic scenarios such as our scenario, which is drone detection and tracking. DL models learn the appropriate features from data automatically while training, as opposed to conventional data-driven approaches which are based on hand-crafted features. This is significant in the UAV detection system since drones come with different characteristics and capabilities [16].

One of the most important components of deep learning for object detection is the Convolutional Neural Networks (CNNs). CNNs are designed to learn automatically and adaptively spatial feature hierarchies from images. This includes learning to detect shapes, edges and motion patterns that characterize drones. Neural Networks are the backbone of the most modern detection algorithms. Some commonly used architectures are utilized in object detection based on deep learning:

- ➔ Faster Region-based Convolutional Neural Networks (R-CNNs) is a two-stage detector that produces and classifies proposals. They are very accurate but slower to draw conclusions, so it is not as good for real-time drone tracking applications.
- ➔ Single Shot Detector (SSD) is a one-stage detector that predicts bounding boxes and class probabilities in a single pass and is very fast and accurate.
- ➔ YOLO is the best real-time object detection algorithm (You Only Look Once), as it detects the objects in a single pass of the Neural Network. It can also detect drones at high frame rates, which makes it ideal for our UAV detection system. YOLO is described with more details in Section 2.4.2.

Another critical process in training process of deep learning detection models is labeling. Labeling involves identifying images with the position and type of objects, in this case, the drones. For example, a labeled dataset will have bounding boxes around UAVs with classes that indicate their type. This process is important as we want to guarantee that the model learns to detect and understand the differences between various drone types appropriately. Also, knowing the detected drone's type is helpful in order to make some extra calculation and measurements we might need.

This is an example of labeling app labelimg:



2.4.2 YOLO for Drone Detection

YOLO is a fast and efficient real-time object detection algorithm that processes the entire image in a single neural network pass, while also predicting bounding boxes and class probabilities across different areas of the image. Additionally, it is extremely useful for real-time systems that have to do with surveillance, drone tracking, and autonomous vehicles.

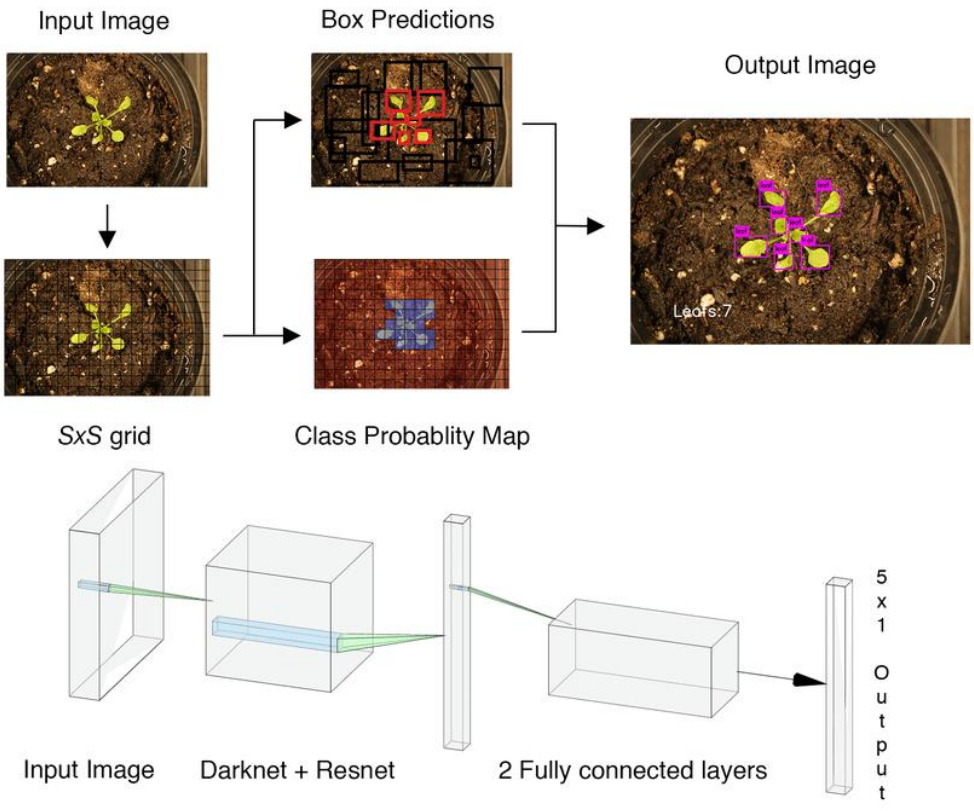
There are several reasons that make YOLO ideal for drone detection systems [17, 18]:

1. Most visual detection systems have a problem when it comes to harsh weather conditions, the similarity of the backgrounds with the drone's color and drone's size and high speed in general. YOLO is trained to generalize well in such situations.
2. Furthermore, it offers much higher detection speeds than other architectures, which is essential for real-time monitoring.
3. YOLO models are able to make multi-drone detections which are extremely helpful for such systems.

4. Most of YOLOs versions are suitable for devices that are low-powered like the NVIDIA Jetson Xavier that we will be using for the implementation of the thesis.

2.4.2.1 Simple Step-by-Step process of how YOLO Works

- ➔ **Image Division:** The input image is getting divided by YOLO into an S x S grid and each of the grid's cells is responsible for detecting objects whose center get inside that cell.
- ➔ **Bounding Box Prediction:** There are 2 or 3 bounding boxes for each cell and each box includes (x, y) which indicates the center of the box and (w, h) which indicates its width and height. Additionally, the boxes include a confidence score which basically tells us how sure the model is that an object is located inside of that box.
- ➔ **Class Probabilities:** These cells also predict a number of conditional class probabilities like drones, person, car. These probabilities indicate whether an object belongs to a certain class.
- ➔ **Predictions Combination:** The combination of the confidence score and the class probabilities adds up the final confidence score which is basically their multiplication.
- ➔ **Non-Maximum Suppression (NMS):** This is used to avoid having duplicate boxes by keeping just the best bounding box. NMS keeps the ones with the highest confidence and removing others with high IoU (Intersection over Union).



➔ **Final Output:** The boxes are being drawn in real-time, and the results are (x, y, w, h), class label and confidence score.

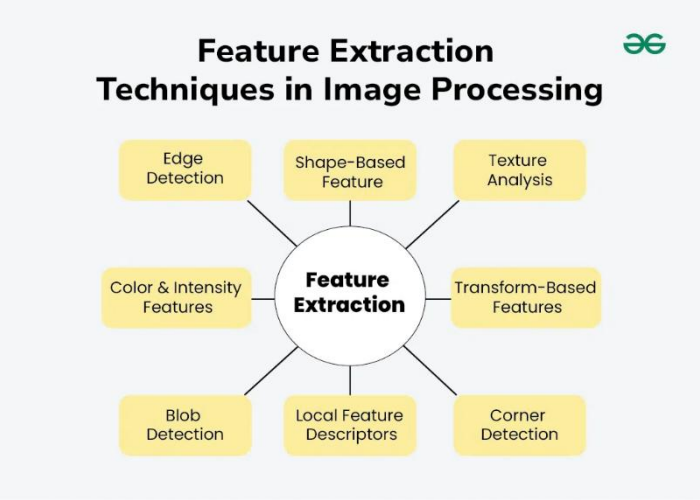
2.4.2.2 YOLO Limitations

YOLO may be very useful and perhaps the ideal algorithm for drone detection systems, but like every other algorithm it comes with its limitations. There is a high probability that a system implemented with YOLO will have difficulties in detecting small drones at long distances due to resolution constraints. Additionally, a large dataset is essential in order to take full advantage of YOLO's capabilities. If the dataset we use is not ideal, our system will lack accuracy and we will have a large number of false positives, as there will be interaction with other objects like, for example, birds. Another limitation that YOLO has is the fact that it doesn't provide any tracking capabilities, which is essential for a drone detection system. We can fix this limitation by adding a tracking algorithm. Such algorithms will be examined in greater detail in section 2.5.[19, 20]

2.4.3 Feature-Based Detection

Feature-Based Detection [21] is able to identify objects based on the analysis of their unique characteristics from images. These detection methods are a basic approach in computer vision, and they are particularly important in UAV detection systems as they provide advanced capabilities when it comes to harsh environments. Some of the key aspects of feature-based detection are:

➔ **Feature Extraction:** For feature extraction, which is basically the detection and description of features within an image, it uses various techniques like SIFT, SURF and ORB. These techniques ensure that the features remain consistent despite any changes in scale or rotation.



➔ **Feature Matching:** After the completion of feature extraction, the algorithm will try to find potential matches of the features that have been extracted and the known features of the dataset.

➔ **Object Recognition:** Lastly, based on matching features, the system decides whether there are any drones present, and if so, it provides their identity.[21]

2.5 Multi-Drone Tracking

In this chapter, the topic of multi-drone tracking is explored, including the challenges it presents and a review of notable tracking algorithms that have been implemented. First of all, understanding what multi-drone tracking is crucial. Multi-drone tracking is the process of detecting, tracking down and identifying multiple drones at the same time. This feature is very important for drone detection systems as it enables non-stop monitoring of UAVs, something that is crucial for airports and military zones. Also, it improves the accuracy of the detections as it maintains the identity of the drones to reduce false positives. Some additional benefits of multi-drone tracking for drone detection systems is that it provides enhanced situational awareness, real-time threat assessment, much better resource allocation, accurate behavior analysis, scalability and historical data logging.

2.5.1 Challenges

Despite the benefits of multiple drone surveillance, there are some challenges that need to be addressed. Here is a list of challenges and how they affect the system:

1. When the drones get behind other objects like trees and building. This causes the system to temporarily lose the drones and their ID. This can be fixed by adding predictive algorithms in the system like Kalman Filters.
2. Most drones have a very similar appearance and sometimes it is quite difficult to distinguish one from another. In order to fix this, there is the need to create a system that will rely on motion history to maintain accurate IDs.
3. Drones speed increased over the years, and it is much harder to detect them frame-by-frame. Also because of their small size it is difficult to detect them from far distances.

In general, we can prevent these problems by AI-based fusion, which is basically the combination of RGB and Thermal cameras in order to have better detection and tracking accuracy.

2.5.2 Tracking Algorithms

To maintain consistent identification of drones across video frames, tracking algorithms are required. These algorithms are responsible for the prediction of UAVs position and handling abstractions. This section discusses in further the two most popular tracking algorithms, Kalman filters and SORT.

2.5.2.1 Kalman Filters

Kalman Filters use the current state of an object to estimate its future position. The future position estimation is the first phase of the algorithm, which is not always right, so there is also a second phase, correction, which updates the next position by using new measurements. This algorithm is ideal for linear motion and enables tracking when the drone is hidden behind other objects or go out of the field of view. In addition, Kalman Filters are used to make the drone's movement paths smoother.

The only limitation of Kalman filters is that if they are used alone in a system, there will be a problem with the association of the data, as this algorithm can't determine which detection corresponds to each drone and that why in most cases, they are used with a matching algorithm called IoU.

2.5.2.2 SORT

Simple Online and Realtime Tracker or SORT is one of the most well-known objects tracking algorithms that enables real-time multi-target tracking by employing two very effective methods: the Hungarian Algorithm and the Kalman Filter. It extrapolates the object's future location based on the current movement using the Kalman Filter, and the Hungarian Algorithm performs the optimal association between the new detections and the extrapolated tracks. They combine SORT, both effective and viable for applications like drone detection where tracking must both happen fast and precisely.

SORT is a great choice for our application on low-power edge boards like the NVIDIA Jetson Xavier because it supports Python 3.6+ and is also very popular among embedded systems because it does not require intensive GPU computing or large models.

However, just as there exists any tracking system, there also exist limitations in SORT. Some of the notable limitations include that the system does not include appearance characteristics such as color, texture, or drone model for discrimination. Thus, when two drones overlap or have the same appearance, the algorithm will switch their IDs. Secondly, the discontinuous motion characteristics that are characteristic in UAVs result in failures in tracking since the Kalman Filters assume smooth and linear motion. Lastly, the forgetfulness about appearance history or long-term IDs makes the system vulnerable to swapping IDs when the detections are temporarily lost or occluded.

One limitation is that SORT is frame-rate dependent: it presupposes that the frame rate has to be constant and high. In low-FPS rates, it might not be capable of keeping constant tracking.

Some of these challenges have been addressed by suggesting more sophisticated variants such as Deep SORT and ByteTrack. Deep SORT employs appearance descriptors such as CNN features for appearance-based object matching that is more robust to occlusions and identity

switching. ByteTrack also enhances detection association reasoning, particularly in the scenario of missed detections or detection confidence loss.

2.5.3 Role of Deque

In tracking and detection of drones, there is a need not only to know that a drone is tracked within one frame, but also where the drone is headed and how it's heading there. That's where Python's deque (double-end queue) proves useful. It's one of the collection modules that's a data structure offering fast and easy addition and removal at both the front and the rear of the queue. Sounds simple enough, yet for tracking systems, very useful.

A deque for every tracked drone in our system is maintained to maintain a record of its past positions in frames. That is, the coordinates for the center of every drone that is detected are appended to the deque for the tracking ID. Over time, this creates a visual trail that is indicative of the drone's movement. Such trails are rendered on the video output so that the operator has a better understanding regarding the drone's path and activity.

A good thing about the use of the deque is that it's memory-friendly. It can define the maximum stored per location number (i.e., 50), and after that maximum is reached, the oldest one gets dropped when new ones are pushed. That keeps the system light and does not push the system into the realm where the system starts running out of memory, which is essential if the system runs on an embedded device like the NVIDIA Jetson Xavier.

Apart from visualizing the path, the history of positions also finds application in the prediction of motion. If the drone gets temporarily occluded (e.g., flying behind a house or tree), the system still knows where the re-appearance most likely would take place by following the earlier motion, thanks to the stored points in the deque. Briefly speaking, deque presents the beautiful and efficient means to manage motion history for all the drones within resource-friendly manner that makes visualization smoother and tracking more robust in real-time scenarios.

2.6 Existing Drone Detection Systems

As the cost and accessibility of drones have decreased, so has the risk that UAVs would penetrate into restricted airspace. As such, both government agencies and private companies have developed advanced drone detection systems. These systems have the capability to identify, trace, and even take down drones using combinations of the following technologies: Radar, RF analysis, machine vision, and machine learning. Some are intended for commercial clients like stadiums or airports, and others are exclusively for the military or the defense sector. The following sections provide a brief description of such systems.

2.6.1 Commercial Systems

Dedrone

Dedrone is among the leading commercial drone detection systems. It unites the application of RF sensors, video cameras, and machine-learning-based software for the detection and identification of the drones and the pilots. It has the capability of tracking rogue drones and sending real-time alerts to the officers. It also possesses one very significant feature that increases the detection with the elapse of time by separating the drones from other aerial objects like birds. Dedrone has been installed at the airports, the prisons, the public events, and critical infrastructure in over 30 countries.

AeroScope by DJI

One such commercial drone detection system is DJI's AeroScope, which is aimed at the targeted DJI drones. It searches for the DJI drones' telemetry signals and is able to locate the operator and the drone. It's being used in stadiums, power plants, and airports worldwide. The limitation being that it only works with UAVs from DJI, it's still feasible since DJI is the leader in the consumer drone market.

2.6.2 Military and Government Systems

Anduril Industries - Lattice AI and Anvil

Anduril developed the Lattice drone-defense system using machine learning-based artificial intelligence, radar, and thermal detection for the identification and tracking of drones. It is augmented by physical interceptor drone called Anvil that kills hostile UAVs by itself and brings them down. Already the U.S. military implemented the system in base protection and field deployment in its arsenal. Lattice boasts the world's first-ever autonomous detection-to-response chain that makes the system one among the most sophisticated drone counter systems in the marketplace.

AUDS (Anti-UAS Defense System)

It was created in collaboration by Enterprise Control Systems, Chess Dynamics, and Blighter Surveillance Systems. It has been deployed by the armed forces worldwide, including the U.S. and the UK. The system utilizes radar, electro-optical tracking, and jamming via RF frequency. It detects the drones, tracks them, and also jams them. It is very mobile and has been deployed in the field as well as for the protection of the armed bases against drones.

Chapter 3

Previous System

This chapter focusses on the detailed description and examination of the previous system's implementation, a drone detection and tracking system, which was the basis for the implementation of the new upgraded counter-drone detection system made for this thesis. The attention of the following sections is directed toward its architecture, key components, and operational workflow. The analysis aims to provide insights into the system's capabilities and identify areas for improvement.

3.1 System Overview

The system was created by Rafael Makrigiorgis in 2020 and had the ability to detect and track unmanned aerial vehicles (UAVs) in real-time, responding to the new requirement for effective security and surveillance procedures to neutralize malicious drone activity. Through the use of the combination of computer vision algorithms and sensor fusion techniques, the system was designed to facilitate timely and accurate detection and tracking of drones within a surveillance region. Some of its aims:

- **Real-Time Detection:** Achieve real-time detection of invading drones into the region.
- **Accurate Monitoring:** Keep monitoring detected drones continuously, recording their activity and paths.
- **Data Integration:** Integrate visual data from cameras with telemetry data to enhance the reliability of detection.
- **User Interface:** Provide operators with a straightforward, easy-to-read presentation of detection and track data to enable decision-making.

3.2 System Architecture

The previous version of the system was built on a modular architecture, which was made up of different components that were combined to detect and track drones. The kind of architecture was simple to scale and maintain if needed.

The system was separated into different modules, every one of them having a specific function to serve in the process of the drone detection system. This allowed developers to build isolated components without altering the system as a whole, making it simpler to maintain and simpler to debug and update. Here is a detailed description of the modules of the previous system:

- ➔ **Input Acquisition Module**: This module was tasked with acquiring real-time video streams and telemetry data from drones. Through subscribing to particular data streams, it gave the system real-time data required for accurate detection and tracking.
- ➔ **Object Detection Module:** The module applied the state-of-the-art computer vision algorithms to analyze incoming video streams to identify potential drone objects in the area being monitored. Applying models like YOLOv4-tiny allowed efficient and rapid detection at the expense of speed or accuracy.
- ➔ **Tracking Module:** Following the detection of a drone, the tracking module continued to monitor its trajectory in subsequent frame. Techniques such as Kalman filtering were utilized for position updating and prediction of the drone so that continuous tracking could be carried out regardless of occlusion or temporary loss of visibility.
- ➔ **Sensor Fusion Module:** In an attempt to make the tracking more accurate and reliable, this module fused the output of multiple sensors. By fusing visual data from cameras and telemetry data, the system gained a clearer picture of where the drone was headed and where it was.
- ➔ **Output Visualization Module:** The module provided real-time visual feedback to the operators, displaying the detected and tracked drones in the area being monitored. The user interface was intuitive, with the capacity for quick data interpretation and decision-making.

3.3 Key Components and Functionalities

As seen in section 3.2, the system is composed of different modules, each designed to perform distinct tasks that collectively enable real-time drone detection and tracking. This section will explain these modules, what they are responsible for and their purpose.

3.3.1 Input Acquisition Module

Input Acquisition module (or alternative input collection unit) is responsible for collecting real-time key data for the detection and tracking processes. It mainly collects two types of data, which provide the foundation upon which all other processing modules operate:

- **Video Stream Capture:** High-definition cameras are deployed in carefully selected locations to maximise aerial coverage and minimise blind spots. These cameras transmit a non-stop live image of the observed airspace, forming the visual backbone of the system.
- **Telemetry Integration:** For the subscription to telemetry feeds from UAVs, the system's designer chooses ROS. These feeds include critical data such as GPS position, altitude, speed and orientation, which are vital for accurate tracking and positioning.

3.3.2 Object Detection Module

After receiving input data with the help of the input acquisition module, the system must check if the drones are present in the observed region. That is performed with the help of object detection algorithms. The module employs light yet powerful neural network structure for processing the frames of the observed video and detecting potential drone objects. That is performed by the application of the following optimizations:

- **YOLOv4-tiny-Based Detection:** This model is the core of the module as it offers fast and accurate detections. YOLOv4-tiny is selected on this occasion instead of any other YOLO model because it works really good in low power devices, like the device that run this system, a Jetson Xavier. To be more specific, the model analyzes each video frame to identify objects, generating bounding boxes, class predictions, and confidence scores.
- **Optimized Detection Parameters:** Confidence threshold and NMS are very important for the detection system as they need to be fine-tuned to balance sensitivity and specificity. We need to be very careful when changing those parameters as false positives and false negatives might appear if they are set incorrectly.

3.3.3 Tracking Module

After detecting a drone, the system needs to continuously monitor its location over time. The tracking module, as the name says, is responsible for tracking the movement of each detected drone, ensuring consistent identification even if the drone moves or temporarily disappears from view. In this particular system two methods are being used:

- **Kalman Filter Tracking:** It is responsible for predicting and updating the positions of the drone over time. Kalman filters take noise into account when it comes to measurements and is one of the most well-known and effective ways used for motion estimation.

- **Object Tracking Association:** The system uses Intersection of Union to match new detections to existing tracks and in order to be able to identify the multiple drones and not confuse them. Also, IoU avoids switching or loss of tracking.

3.3.4 Sensor Fusion Module

So far, the system rely only on visual data, something which isn't ideal for such important systems. In order to have more accurate drone localizations, this module combines data from different sources. The main sources are:

- **Visual-Telemetry:** The system converts frame-level pixel positions into geographic coordinates. This conversion happens because of the combination of image-based detection data and telemetry data, and it also helps pinpoint drone locations with greater spatial accuracy.
- **Temporal Synchronization:** It also ensures that all incoming data streams are synchronized in time. Proper alignment of visual and telemetry inputs is essential to maintain accuracy during sensor fusion and avoid discrepancies.

3.3.5 Output Visualization Module

To assist operators with understanding what is happening real-time, the system provides graphical feedback from the user interface. The module presents all the track and detection information that is relevant in understandable terms and also presents warnings if detection of the drone is present.

- **Real-Time Display Interface**: A tailored GUI overlays the track data on the real-time camera display, forming a graphical interface for users, allowing drone motion to be observed real-time. The interface is easier to
  read and provides instant situation awareness.
- **Alert and Notification System:** The module possesses alert functions, both auditory and visual, that are triggered when drones are detected. The alerts are designed to draw immediate attention to the potential threats, in support of prompt decision-making.

3.4 Technical Implementation Details

The following subsections focus on the technical and software part of the system. It reports about what libraries were included in the code and how in general the system was assembled.

3.4.1 Programming Languages and Libraries

The system was implemented using Python. As I have been told, the developer of the system was between two languages. Python and C++. He chose Python for its flexibility, readability, extensive support for computer vision and robotics tasks and in general because C++ was way more complicated for the implementation of such system (also C++ would have been better for performance). The main libraries and frameworks used include:

- ➔ **OpenCV:** Used for frame capturing, video processing, displaying results, drawing bounding boxes, and managing GUI windows.
- ➔ **NumPy:** Used for handling matrix operations and array manipulations, especially in Kalman filtering and coordinating transformations.
- ➔ **PyTourch / OpenCV DNN:** Used optionally for loading and running the YOLOv4-tiny neural network. The system supports both Torch-based and OpenCV DNN-based inference engines.
- ➔ **ROS:** This actually provides middleware for real-time telemetry communication via topics and custom messages (std_msgs, Float64MultiArray, kios.msg.Telemetry).
- ➔ **Custom Modules:** Custom modules such as detector, Kalman, CRPS, and queue thread are being used, each one of them having its own special role and responsibility on the system.

3.4.2 Real-Time Data Handling

- ➔ **Video Stream Input:** The system reads from an RTMP video stream **("rtmp://192.168.10.44/live/")**, capturing live footage from a surveillance camera or drone feed, in our case from another drone's feed. There is also support for reading local video files which is basically used for practicing purposes.
- ➔ **Frame Processing Thread:** A separate thread (VideoStream) handles frame acquisition, which allows continuous reading of video data while the main thread performs detection and tracking. This is a crucial part of the system.
- ➔ **Telemetry Input:** In order to collect real-time telemetry, the system uses two ROS subscriptions. The matrice300/EKF, which provides estimated position values used in CRPS updates and the matrice300/Telemetry which captures GPS coordinates, heading, etc., from the UAV.

3.4.3 Neural Network Detection

In this system, the model that has been used is YOLOv4-tiny and it is chosen because of its lightweight nature, capable for deployment on resource-constrained edge devices like the NVIDIA Jetson Xavier. In order to run the system optimally though, the resolution needs to be

set at 416x416, something which is not happening in this system, and this is one of the main reasons that the system is slow. Also, the threshold being used are Confidence = 0.3, IOU = 0.3 and NMS = 0.2. Additionally, the system optionally enables GPU acceleration (gpu=True) for OpenCV DNN inference, leveraging Jetson's CUDA cores.

3.4.4 Tracking and Filtering

As mentioned before, for tracking and filtering the developer chose Kalman filter and Object association using IoU in order to achieve optimal results.

3.4.5 Sensor Fusion and Localization

For sensor fusion and localization, there are two main ideas behind them:

- ➔ **CRPS Module:** Fuses telemetry (GPS) and vision data to convert bounding box locations (pixel-based) into real-world GPS coordinates. This is especially useful when drones are far or moving erratically.
- ➔ **Temporal Synchronization:** The system assumes telemetry and visual frames arrive in sync, updating CRPS only when both streams have data.

3.4.6 Execution Flow Summary

To sum up the above, the system works as follow:

1. Initialize ROS node and subscribers.
2. Initialize video stream and YOLO detector.
3. Begin frame acquisition loop.
4. For each frame:
    a. Detect drones using YOLO.
    b. Update CRPS with the latest telemetry.
    c. Initialize or update tracker (Kalman + IoU).
    d. Overlay detection and tracking results.
    e. Save frame to output video (optional).
    f. Display frame on screen.
5. Terminate when the user presses q or ESC.

3.5 Limitations and Challenges of the Previous System

The initial drone detection and tracking system has some limitations that need to be addressed which affect the accuracy and performance. In this section some of these limitations will be mentioned.

- There is a limited generalization across drone types, as the model hasn't been trained for different drone types but just for the DJI Matrice-300 RTK. This affects the accuracy of the system as it will not be able to detect other drone types. This makes the system useful for general purposes other than the detection of DJIs Matrice-300.
- There is a relatively small and weak dataset of drone images as almost all the images collected have as background the sky which will make it very strong to detect drones in the sky but simultaneously impossible to detect drones in other backgrounds such as building and trees because the model is not trained to do so.
- There is a lack of drone information as it doesn't classify the drones, it doesn't calculate its distance nor its speed from the area. It only tell us that there is a drone somewhere in the area.
- Because of the way the system is implemented there are performance bottlenecks, as the Jetson cannot keep up with the real-time data. The system was probably not implemented specifically for Jetson and that is a huge setback as Jetson Xavier is the device responsible for running the system. Knowing that, we must make sure that the system is as lightweight as possible in order to make it work on the Jetson.

Chapter 4

Implementation and Methodology

This chapter gives emphasis on the drone detection and tracking system's implementation, methodology and technical architecture. The following sections of chapter 4 will examine both versions of the system which are the main system which works on embedded devices like Nvidia Jetson Xavier and the version that uses videos to make detections on a normal PC. Both of these systems are designed to detect, identify, track and collect information for the drones using computer vision and other machine learning techniques. In addition, every module, class and function of the systems code will be analyzed in order to get an idea of how the system was created and why those modules are important for a detection system. The architecture of the modules of the system works as shown in the diagram below.

4.1 Architecture Diagram

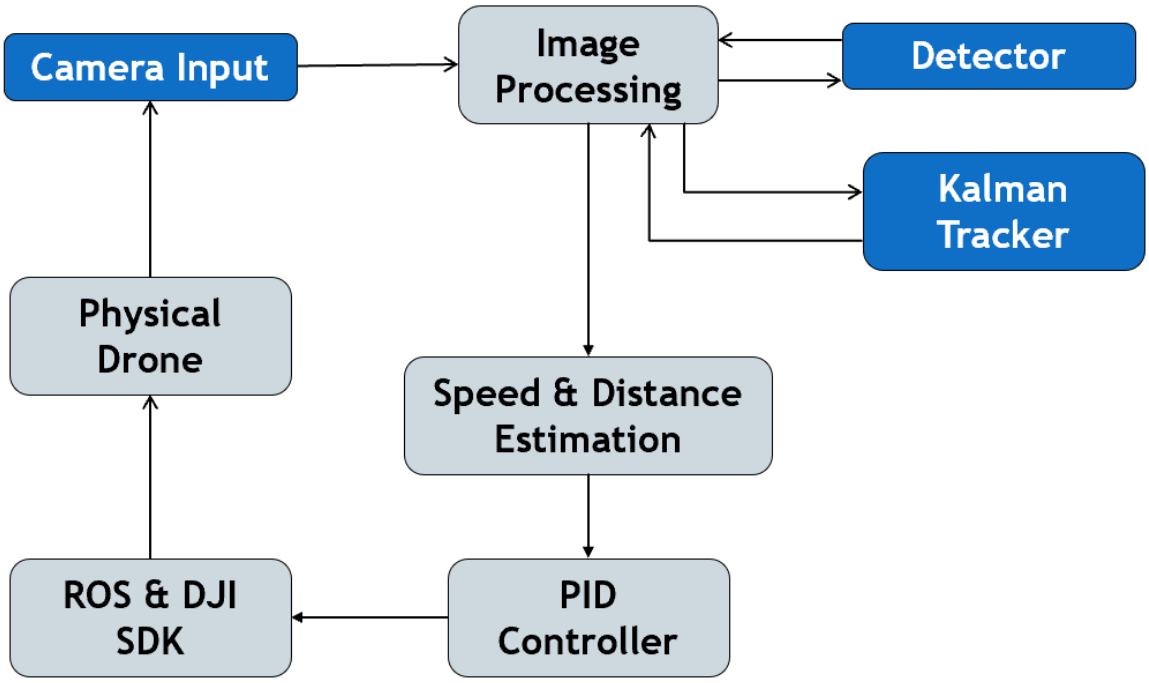
The architecture of the modules of the system works as follows:

- ➔ We have the input from the camera or the pre-recorded video accordingly, which is converted with the help of CvBridge into OpenCV format, so that this input can be given to Image Processing.
- ➔ In Image Processing now, the data is processed, where it is connected to two sub-modules. The Detector, where the image will be given to see if there is an object that

satisfies the model and will return bounding box, confidence score, drone type. The Kalman Tracker, where a unique ID will be given to the detection and it will be tracked.

➔ Subsequently there will be speed and distance calculation.

➔ And then angle correction calculation with the PID controller, which with the help of ROS will send any commands to the drone.



4.2 Shared Detection Module

Both embedded system and PC-based systems use a common class, the detector class. This particular class wraps the YOLO drone detection framework with the help of OpenCV and, to be more exact, OpenCV's Deep Neural Network module. The purpose of this class is to load the neural network weights we generated during the training of the system with the help of Darknet, the configuration file, and the file with the class names. Once those files are loaded, the class will perform inference, and it will return structured detection data. The class is named **detection.py**

4.2.1 Initialization and Configuration

```
        self.net = cv2.dnn.readNet(self.weights, self.config)
```

➔ In the following line of code, we initialize the detector using the selected weights and configuration file.

```
        if self.gpu:
            print("[INFO] Using CUDA for DNN inference (if supported).")
```

```
            self.net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
            try:
                self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA_FP16)
            except:
                self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
        else:
            self.net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
            self.net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)
```

➔ As in most systems, speed and performance is key. In order to run the interface with optimal performance, the use of the GPU is necessary. Depending on the following gpu flag, the model either runs on CPU or GPU. If the gpu flag is true, meaning there is a GPU available, the system sets CUDA as the backend for the DNN module. In addition to that, the system checks if FP16 (16-bit floating point) precision is supported. This must be checked because the 16-bit floating point is even faster than the regular 32-bit and uses less memory. If the gpu flag is false, then the network will run on the CPU using OpenCV's default backend.

➔ GPU acceleration via CUDA is critical for real-time inference, especially on embedded hardware like the Nvidia Jetson.

4.2.2 Detection Method

This function detects all the objects that appear in a frame, returning bounding boxes, confidence scores, and class identifiers. These are then used for identification, classification and further analysis. The purpose of it is to provide a fast and accurate drone detection backend, which will be useful for both systems.

```
    def detect(self, frame):
        imH, imW = frame.shape[:2]
        try:
            class_ids, confidences, boxes = self.model.detect(
                frame,
                confThreshold=self.conf_thresh,
                nmsThreshold=self.nms_thresh
            )
        except Exception as e:
            print("[ERROR] Exception during detection:", e)
            return [], [], [], []

        if len(boxes) == 0:
            return [], [], [], []
```

```
        class_ids   = class_ids.tolist()
        confidences = confidences.tolist()
        boxes       = boxes.tolist()

        indices = [[i] for i in range(len(boxes))]

        return indices, boxes, class_ids, confidences
```

The following is a brief explanation of how the code works:

- ➔ The system extracts the height and width of the image, not the channels (that's the purpose of [:2].
- ➔ Then it runs the object detection by the following function self.model.detect(...), which returns a list of detected objects class numbers (class_ids), the confidence of the detection and the bounding boxes.
- ➔ Lastly, to avoid any errors we convert the 3 things that the detector returns to lists, and we return 4 things:
  - indices: position index of each detection
  - boxes: list of [x, y, w, h] bounding boxes
  - class_ids: corresponding object type for each box
  - confidence: how confident the model is for each detection
- ➔ This method safely and efficiently performs drone detection in an image using a model based on YOLOv4. This method was chosen for the drone detection system because it ensures error handling during detection, consistency of return types and robust output even when there are no drones/objects detected. The only limitation is that the detection accuracy depends heavily on the training dataset, as it need well-trained weights. Another small setback is that it has very high computational loads without the use of GPU.

4.3 External Libraries and Frameworks

This project utilized some of the more popular libraries to enable computer vision, control, and communications functionality. The following is a breakdown of the primary libraries used and what it was used for

- ➔ **OpenCV (cv2):** Used for drawing boxes, displaying detections, and printing annotated video streams. Also used for performing YOLOv4 inference by utilizing the dNN module.

- ➔ **rospy:** Python ROS client library, used in subscribing from image topics, publishing control commands, and system-level module management.
- ➔ **numpy**: Made fast numerical operations, most notably used in Kalman filtering, speed and distance computations, and array operations.
- ➔ **cv_bridge:** ROS Image Messages were converted into OpenCV structures in order to process the frames through the detection pipeline.
- ➔ **dji_sdk:** Made the system able to talk with DJI drones, controlling flight control and publishing joystick control.
- ➔ **geometry_msgs & sensor_msgs:** Simple ROS message types for publishing motion instructions (Twist) and subscribing to images (Image).
- ➔ **Json:** Used to read metadata such as drone sizes from configuration files.
- ➔ **math & collections.deque:** Also used in geometric computations, estimating velocity, and keeping track of recent drone locations in path recordings.

These libraries, used in combination, offered the detection, monitoring, control, and visualization components of the system while ensuring compatibility for both embedded and PC-based implementations.

4.4 Utility Functions

The drone_util.py class contains some key functions for geometric and mathematical calculations used throughout the system. This section will explain what each function do, list their purpose and where they are used.

4.4.1 Focal Length Computation

```
def compute_focal_length(image_width, horizontal_fov_deg):
    half_angle_rad = math.radians(horizontal_fov_deg / 2)
    return (image_width / 2) / math.tan(half_angle_rad)
```

- ➔ This function is used to calculate the camera's focal length in pixels based on the image's width (resolution, in our case we use 640x640 for performance purposes) and the horizontal field of view (FOV).
- ➔ In order to calculate it, the system converts the horizontal FOV to radians and then applies the pinhole camera model formula. f=tan(horizontal_fov/2)image_width/2
- ➔ This will return how many pixels is the focal length of the camera and we will need it in order to calculate the distance of the drone from the camera.

4.4.2 Intersection of Union calculation

```
def iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])
    yA = max(boxA[1], boxB[1])
    xB = min(boxA[2], boxB[2])
    yB = min(boxA[3], boxB[3])
    interW = max(0, xB - xA)
    interH = max(0, yB - yA)
    interArea = interW * interH
    boxAArea = (boxA[2] - boxA[0]) * (boxA[3] - boxA[1])
    boxBArea = (boxB[2] - boxB[0]) * (boxB[3] - boxB[1])
    if boxAArea + boxBArea - interArea == 0:
        return 0.0
    return float(interArea) / float(boxAArea + boxBArea - interArea)
```

- ➔ The iou function calculates the Intersection of Union between 2 bboxes which is basically a metric used to determine how similar or overlapping two boxes are.
- ➔ In this function we find the overlapping area between box A and box B, and we compute the union of the two boxes in order to be able to find and return the IoU which comes from the following formula: IoU=Union Area/Overlapping Area.
- ➔ This is commonly used in tracking or matching detections, especially when assigning detections to tracks in object tracking systems.

4.4.3 Bounding Box Format Conversion

```
def convert_xywh_to_xyxy(x, y, w, h):
    return (x, y, x + w, y + h)
```

- ➔ This function is used to convert the format of the bounding boxes from (x,y,w,h) which is top-left corner, width and height, to (x1,y1,x2,y2) which is the coordinates of the top-left and bottom-right corners.
- ➔ The reason for that conversion is that most algorithms are expecting the bounding boxes to have this format.

4.4.4 Group Center Computation

```
def compute_group_center(detections):
    if not detections:
        return None
    x_min = min(det['bbox'][0] for det in detections)
    y_min = min(det['bbox'][1] for det in detections)
```

```
    x_max = max(det['bbox'][2] for det in detections)
    y_max = max(det['bbox'][3] for det in detections)
    return ((x_min + x_max) / 2.0, (y_min + y_max) / 2.0)
```

- ➔ This function calculates the center point of a group of detected drones, and we need it to adjust the positioning of the drone in such a way that it will be able to have all the detected drones in its FOV.
- ➔ In order to have correct results, we need to find the min and max coordinates of all the detections in order to calculate the center which comes from the following formula: centerx=(xmin+xmax)/2, centery=(ymin+ymax)/2

4.5 PID Control

The class pid_controller.py implements a simple PID (Proportional-Integral-Derivative) controller which is used for used for yaw control of the drone. The yaw refers to the rotation of the drone across the vertical axis. The system should change the yaw based on the position of the drone detected, as the FOV of the drone's camera must always be aligned with the object being detected. In short, the position of the detected drone will be obtained, and yaw commands will be sent, if necessary, in order to rotate the drone and center the detected drone in the FOV. Let's take a closer look at how the PID controller works in our system and what it is in general.

- ➔ In general, a PID controller calculates continuously an error value as the difference between a desired setpoint and a measured value and applies a correction based on three components. Those components are the present error (Proportional), the accumulated past error (Integral) and the rate of error change (Derivative).
- ➔ The goal is to bring the system to the target value smoothly and efficiently, without oscillations or overshoot.

```
class PIDController:
    def __init__(self, kp, ki, kd, setpoint=0.0):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.setpoint = setpoint
        self.last_error = 0.0
        self.integral = 0.0

    def update(self, measurement, dt):
        error = self.setpoint - measurement
        self.integral += error * dt
        derivative = (error - self.last_error) / dt if dt > 0 else 0.0
```

```
        output = self.kp * error + self.ki * self.integral + self.kd * derivative
        self.last_error = error
        return output
```

➔ The parameters of the constructor are:

- kp - proportional gain.
- ki - integral gain.
- kd – derivative gain.
- setpoint – which is the desired target value.
- last_error - keeps track of the previous error for derivative calculation.
- integral - accumulates error over time for the integral term.

➔ By adjusting those parameters, you can control how aggressively the system reacts to errors.

➔ Update function calculates the new control output based on the current observed value and the time step since the last update. This function works with the following steps:

- Determines how off is the current value from the desired one (error = self.setpoint – measurement).
- Accumulates the error over time, useful for eliminating steady-state bias (self.integral += error * dt).
- It measures how fast the error is changing and helps to avoid overshooting (derivative = (error - self.last_error) / dt if dt > 0 else 0.0).\
- Combines all the three terms into a single control signal (output = self.kp * error + self.ki * self.integral + self.kd * derivative).
- Stores the current error to compute the next derivative (self.last_error = error).

➔ Note that the PID Controller implemented on the system can only control yaw and not the full motion of the drone.

➔ This is how the system will work with the PID and Yaw adjustment integration. The red crosses represents the center of the image and the green cross the center of the drone detected. The system will calculate the estimated error (white line, difference from the two centers) and it will send a joystick yaw command through DJI SDK, and it will rotate the drone. In the case of the image, it will send a command to rotate left.

4.6 Kalman Filter-Based Tracking

The kalman_tracker.py (KalmanTracker class) contains the magic to ensure frame continuity through smart matching of detections in a manner befitting over time. It ensures that all detected drones receive persistent labelling and were tracked even upon temporary failure in detection or upon occlusions. Using a prediction strategy, the tracker neither relies on the visibility of the object in the current frame alone but rather projects where it should appear in the next frame and so infuses much needed robustness and stability to the system. It becomes extremely crucial in cases where dynamic scenes have drones behaving in a wayward manner or becoming entangled with environmental obstacles.

Moreover, the system remembers the past tracked drones and hence can reinvent identities should a lost drone resurface. In effect, the class acts to be the glue that holds detection points and generates worthwhile tracks from where higher-level activities such as behavior analysis, motion predictions, and responsive command can be derived.

In the following subsections there will be a presentation of the functions of the class, what their purpose is and how they work.

4.6.1 Tracker Initialization

```
def __init__(self, iou_threshold=0.5, max_inactive=150):
    self.iou_threshold = iou_threshold
    self.max_inactive = max_inactive
    self.track_info = {}
    self.lost_tracks = {}
    self.next_track_id = 1
    self.available_ids = []
```

➔ This is the initialization of the tracker, which take as parameters the iou_threashold, which is the minimum IoU required to match a detection with a track and the max_inactive, which is the maximum number of frames a track can go unmatched before it is removed.

➔ The variables that are initialized on it are: track_info, which represents the active tracks (the currently tracked drones), the lost_tracks, which represents the recently lost tracks,

the next_track_id, which is the ID that the tracker will assign to a new track and the available_ids, which are the IDs from removed tracks.

4.6.2 Kalman Filter Creation

```
def create_kalman_filter(self, center):
        kf = cv2.KalmanFilter(4, 2)
        kf.transitionMatrix = np.array([1, 0, 1, 0],
                                       [0, 1, 0, 1],
                                       [0, 0, 1, 0],
                                       [0, 0, 0, 1]], np.float32)
        kf.measurementMatrix = np.array([1, 0, 0, 0],
                                        [0, 1, 0, 0]], np.float32)
        kf.processNoiseCov = np.eye(4, dtype=np.float32) * 0.03
        kf.measurementNoiseCov = np.eye(2, dtype=np.float32) * 0.5
        kf.errorCovPost = np.eye(4, dtype=np.float32)
        kf.statePost = np.array([[np.float32(center[0])],
                                 [np.float32(center[1])],
                                 [0],
                                 [0]], np.float32)
        return kf
```

➔ The create_kalman_filter function is creating and initializing a Kalman Filter which will be centered with the detected drone's center.

➔ This function is crucial for the tracking methodology of the system as Kalman filters are used to predict the next state of the drone and to correct this prediction when new data are available. Let's see what the purpose of each variable is:

- The **transitionMatrix** defines how the object's state (position and velocity) evolves over time.
- The **measurementMatrix** specifies what aspects of the state are directly observable, in this case, the position coordinates.
- The **processNoiseCov** accounts for uncertainty in the drone's movement, representing the noise in the prediction model.
- The **measurementNoiseCov** quantifies noise in data measured from the detector

➔ These four matrices together allow the Kalman Filter to have a strong estimate about the position of the drone even if the observations get noisy or lost momentarily.

4.6.3 New Track Initialization

This function with the name of _init_track() have the responsibility to initialize a new object track in the system whenever a detection can't be associated with confidence with an existing track. This function is very important when it comes to the first appearance of a drone or when the system needs to re-identify a previously lost drone.

```
    def _init_track(self, detection):
        x1, y1, x2, y2 = detection['bbox']
        class_id = detection['class_id']
        score = detection['score']
        w = x2 - x1
        h = y2 - y1
        center_x = 0.5 * (x1 + x2)
        center_y = 0.5 * (y1 + y2)
        kf = self.create_kalman_filter((center_x, center_y))
        reid_found = None
        for lost_id, lost_info in self.lost_tracks.items():
            if iou(detection['bbox'], lost_info['bbox']) > 0.5:
                reid_found = lost_id
                break
        if reid_found is not None:
            new_id = reid_found
            del self.lost_tracks[reid_found]
        else:
            if self.available_ids:
                new_id = min(self.available_ids)
                self.available_ids.remove(new_id)
            else:
                new_id = self.next_track_id
                self.next_track_id += 1
        self.track_info[new_id] = {
            'bbox': (x1, y1, x2, y2),
            'pred_bbox': (x1, y1, x2, y2),
            'class_id': class_id,
            'score': score,
            'drone_name': "Unknown",
            'drone_size': 0.96,
            'color': (0, 0, 255),
            'trail': deque(maxlen=50),
            'prev_distance': None,
```

```
            'inactive_count': 0,
            'kalman': kf,
            'size': (w, h),
            'last_update_time': rospy.get_time()
        }
```

- ➔ Firstly, it extracts the bounding box coordinates (x1,y1,x2,y2), the class ID and the confidence score from the detection input. After the extraction, the function calculates the height, the width and the coordinates of the center of the object. These coordinates center_x and center_y will be used to create a new Kalman Filter by calling the appropriate function.
- ➔ The algorithm above will also check if the new detection matches any of the previous lost tracks before creating a new one. This can be done with the help of the intersection of union function which was implemented before. If the iou is higher than 0.5 then the function assumes the object has been re-identified and reuses the existing track ID, otherwise it creates a new one.
- ➔ Once we determine the ID, a new track is stored in the track_info dictionary, which includes the bounding box, the predicted bounding box, information about its class, confidence score, default drone properties, color for visualization, a trail deque for storing the object's movement history, a Kalman Filter instance, and metadata such as the object's size and the timestamp of the last update.
- ➔ The benefits of keeping this data are that the system can track objects over time, estimate speed and distance, and allow smooth visualizations even in the event of temporary occlusions or lost detections. This capability is needed to support persistent and consistent multi-drone tracking in the system.

4.6.4 Update Track

This function corrects the status of an existing track from an incoming detection and is reported when the tracking system successfully matches an incoming detection with an existing drone track. In addition, the function is intended to correct the position of the track from a new observation and make the tracking consistent between images.

```
    def _update_track(self, track_id, detection):
        x1, y1, x2, y2 = detection['bbox']
        class_id = detection['class_id']
        score = detection['score']
        track = self.track_info[track_id]
        w = x2 - x1
        h = y2 - y1
        center_x = 0.5 * (x1 + x2)
```

```
        center_y = 0.5 * (y1 + y2)
        measurement = np.array([[np.float32(center_x)],
[np.float32(center_y)]])
        track['kalman'].correct(measurement)
        track['bbox'] = (x1, y1, x2, y2)
        track['class_id'] = class_id
        track['score'] = score
        track['inactive_count'] = 0
        track['size'] = (w, h)
        track['trail'].append((center_x, center_y))
```

- ➔ This function is very similar to the _init_track() function as they both extract the same information from the detection input and they both calculate height, width and the center's coordinates. The only difference they have is that it retrieves the corresponding track from the track_info dictionary using the track_id.
- ➔ The key part of this process is that it uses the correction step of the Kalman Filter which helps to refine the prediction made by the Kalman Filter using the actual observed position, allowing the filter to stay synchronized with the real-world movement of the object. It also reduces the influence of noise or small inaccuracies in detection.
- ➔ After this crucial part, the function updates the following properties of the track:
  - Bounding Box.
  - Class_id and detection score.
  - It resets the inactive_count.
  - Size of object.
  - Trail deque, which basically stores the object's recent history of movement.
- ➔ The purpose of this function is to ensure that the tracking system maintains accurately, provide updated information about each drone's position, size, classification, and trajectory.

4.6.5 Main Update Function

```
    def update(self, detections):
        for tid, info in self.track_info.items():
            predicted = info['kalman'].predict()
            pred_center = (predicted[0, 0], predicted[1, 0])
            w, h = info['size']
            pred_bbox = (pred_center[0] - w/2, pred_center[1] - h/2,
                         pred_center[0] + w/2, pred_center[1] + h/2)
            info['pred_bbox'] = pred_bbox
            info['inactive_count'] += 1
```

```
        def association_metric(det_bbox, pred_bbox):
            iou_val = iou(det_bbox, pred_bbox)
            det_center = ((det_bbox[0] + det_bbox[2]) / 2.0,
                          (det_bbox[1] + det_bbox[3]) / 2.0)
            pred_center = ((pred_bbox[0] + pred_bbox[2]) / 2.0,
                           (pred_bbox[1] + pred_bbox[3]) / 2.0)
            center_distance = math.sqrt((det_center[0] - pred_center[0])**2 +
                                        (det_center[1] - pred_center[1])**2)
            boost = max(0, (50 - center_distance)) / 50.0
            return iou_val + boost, center_distance

        for det in detections:
            best_metric = -1.0
            best_track_id = None
            for tid, info in self.track_info.items():
                if info['inactive_count'] < self.max_inactive:
                    metric, center_distance = association_metric(det['bbox'],
info['pred_bbox'])
                    if metric > best_metric and (iou(det['bbox'],
info['pred_bbox']) > self.iou_threshold or center_distance < 50):
                        best_metric = metric
                        best_track_id = tid
            if best_track_id is not None:
                self._update_track(best_track_id, det)
            else:
                self._init_track(det)

        stale_ids = [tid for tid, info in self.track_info.items() if
info['inactive_count'] >= self.max_inactive]
        for tid in stale_ids:
            self.lost_tracks[tid] = self.track_info[tid]
            del self.track_info[tid]
        return [(tid, info) for tid, info in self.track_info.items()]
```

➔ The main update function represents the core of logic of the tracking process, in which the functions described in the previous subsections are included and it has the following responsibilities.

- **The Prediction Step:** In this step, the function uses the Klaman Filter for each active track to predict the object's next position in order to maintain tracking no matter of the circumstances.

- **Association of Detections to Tracks:** In this part of the function there is a definition of an ***association_metric()*** which combines Intersection of Union and center point distance between detections and predictions in order to determine the best matching track. Each new detection is either assigned to an existing track by using ***_update_track*** or is used to create a new track by using ***_init_track*** if there is no suitable match.
- **Track Maintenance:** If there are tracks that haven't been updated for a specific number of frames, called max_inactive, then they are considered lost and stored in the lost_tracks dictionary.

4.7 Drone Image Handler

The drone_image_handler class is responsible for capturing and managing image data from the drone's camera in real-time. It acts as the image capture module of the system and the main purpose of this class is to receive live video stream data from the drone's camera through a ROS topic, convert it into an OpenCV-compatible format, then resize it, and lastly to store the latest frame so that it can be used by the detection and tracking components of the system.

```
import cv2
import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class DroneImageHandler:
    def __init__(self, image_topic, image_width, image_height):
        self.bridge = CvBridge()
        self.current_frame = None
        self.image_width = image_width
        self.image_height = image_height
        rospy.Subscriber(image_topic, Image, self.image_callback)
        rospy.loginfo("Subscribed to image topic: {}".format(image_topic))
    def image_callback(self, msg):
        try:
            frame = self.bridge.imgmsg_to_cv2(msg, desired_encoding="bgr8")
            self.current_frame = cv2.resize(frame, (self.image_width,
self.image_height))
        except CvBridgeError as e:
            rospy.logerr("CvBridge Error: {}".format(e))
    def get_frame(self):
        return self.current_frame
```

- ➔ This class contains three main functions, __init__, image_callback() and get_frame()
- ➔ The constructor (__init__) function sets up the subscriber to the image topic using rospy.Subscriber and the image dimensions to ensure consistency.
- ➔ The image_callback() function is automatically activated when a new image is captured, and it uses cv_bridge library to convert the image from ROS format to OpenCV format. This place of the class is also where the resizing happen.
- ➔ The last function, get_frame(), simply returns the latest received and processed image frame.
- ➔ Normalizing the image size beforehand allows it to give a standard input quality, useful to help keep the accuracy of the object detection model consistent. It is also an integral bridge between ROS communication layer and OpenCV-based detection pipeline to facilitate smooth and uninterrupted information flow between system components.

4.8 Drone Controller

The drone_controller.py class handles the control of communication between the DJI drone flight control system and the ROS services of the SDK of DJI. It achieves control authority, sets the joystick control mode, and publishes motion command depending on detection and monitoring results. This provides real-time autonomous control of the pitch and forward flying of the drone in real-time.

This class fulfills five generic roles:

- The procurement of control rights from the DJI SDK so that onboard software may use the drone as and when needed.
- To indicate joystick mode, i.e., how to interpret the motion commands (yaw).
- Print joystick command using ROS services to drive drone movement.
- To transmit Twist messages to visualize movement orders or to debug.
- To relinquish control securely during shutdown of the system, and to restore control to the pilot to avoid accidents.

Here is an explanation of each of the functions implemented in this class to understand how the five rules mentioned above are achieved. In summary, these functions will be __init__(), init_control(), send_command(), shutdown().

```
def __init__(self, agentName):
        self.agentName = agentName
        self.sdkAuth_proxy = None
        self.control_pub = None
```

```
        self.twist_pub =
rospy.Publisher('/dji_sdk/flight_control_setpoint_generic', Twist,
queue_size=10)
        self.init_control()
```

➔ This function initializes the controller with the given drone's name which will be given as a parameter (agentName).

➔ After the initialization, it creates a ROS publisher for Twist messages. A Twist message represents velocity commands for a robot (or in this case, a drone). It is used to control linear and angular motion in space. This thesis requires only the control of the angular motion in order to rotate the drone when needed.

```
def init_control(self):
        rospy.loginfo("Waiting for service:
{}/obtain_release_control_authority".format(self.agentName))
        rospy.wait_for_service(self.agentName +
'/obtain_release_control_authority')
        rospy.loginfo("Waiting for service:
{}/set_joystick_mode".format(self.agentName))
        rospy.wait_for_service(self.agentName + '/set_joystick_mode')

        try:
            self.sdkAuth_proxy = rospy.ServiceProxy(self.agentName +
'/obtain_release_control_authority', ObtainControlAuthority)
            joyMode = rospy.ServiceProxy(self.agentName +
'/set_joystick_mode', SetJoystickMode)
            result = self.sdkAuth_proxy(True)
            rospy.loginfo("Control authority obtained: %s", result)
            result = joyMode(1, 0, 1, 1, 1)
            rospy.loginfo("Joystick mode set: %s", result)
        except rospy.ServiceException as e:
            rospy.logerr("Service Call Failed: %s", e)
        rospy.wait_for_service(self.agentName + '/joystick_action')
        self.control_pub = rospy.ServiceProxy(self.agentName +
'/joystick_action', JoystickAction)
        rospy.sleep(1)
```

➔ This function's purpose is to handle the initial setup required to control the drone using DJI's SDK and it waits for some necessary services to be available. Those services are obtain_release_control_authority, set_joystick_mode, and joystick_action.

- **obtain_release_control_authority** service is responsible to request control authority of the drone. When this service is called with "True" then the system takes

control of the drone, and it gets on autonomous mode. When we call it with “False”, then control is released so the drone can be operated manually again.

- **set_joystick_mode** service configures how the joystick-style commands are interpreted by the drone. It sets the control mode (e.g., attitude, velocity), coordinate frame, and whether inputs like roll, pitch, and yaw are enabled.
- **joystick_action** service is responsible for sending motion commands once the other two services are set. It takes a JoystickParams message which includes values for x, y, z, and yaw to steer the drone accordingly.

```
def send_command(self, forward, yaw):
        joydata = JoystickParams()
        joydata.x = forward
        joydata.y = 0.0
        joydata.z = 0.0
        joydata.yaw = yaw
        try:
            response = self.control_pub.call(joydata)
            rospy.loginfo("Sent joystick command -> Forward: {:.4f}, Yaw:
{:.4f} | Response: {}".format(forward, yaw, response))
        except rospy.ServiceException as e:
            rospy.logerr("Joystick command failed: %s", e)

        twist = Twist()
        twist.linear.x = forward
        twist.angular.z = yaw
        self.twist_pub.publish(twist)
```

➔ This function’s main purpose is to send movement instructions to the drone based on forward and yaw inputs. It does this by calling the DJI joystick control service and by publishing a Twist message for feedback/monitoring.

➔ Inside this function, a JoystickParams message is created with four attributes: x which indicates forward/backward speed, yaw which indicates the drone’s rotation and y,z which are not used in our case.

➔ After the creation, the joystick command is sent using ***self.control_pub.call(joydata)***, which basically tells the DJI drone to move according to the provided x and yaw.

➔ The last action of the function is that a ROS Twist message is created and published to /dji_sdk/flight_control_setpoint_generic, which is often used for visualization, debugging, or logging purposes within ROS tools like rqt_graph or rviz.

```
    def shutdown(self):
        try:
```

```
            self.sdkAuth_proxy(False)
            rospy.loginfo("Released control authority. Manual control is now
enabled.")
        except rospy.ServiceException as e:
            rospy.logerr("Failed to release control authority: %s", e)
```

➔ This function is responsible for safely releasing the drone's control authority when the system is shutting down or no longer needs to control the drone.

➔ It's basically sending a False to the obtain_release_control_authority in order to give the control back to the drone's controller.

4.9 Video Logging

The video_recorder.py class captures video output of the system as it detects drones and stores it in a file locally. It captures all labeled picture data - detected drones, IDs of detected drones, distance, and speed - filtered by the system in real-time while running. Capturing and saving this visual output is essential for post-analysis, debugging, and demonstration of the system's performance. This class encapsulates OpenCV's video writing functionality in a clean and modular manner so it may be easily used in larger programs with drones with ROS.

```
def __init__(self, output_file, image_width, image_height, fps=20.0):
        fourcc = cv2.VideoWriter_fourcc(*'XVID')
        self.writer = cv2.VideoWriter(output_file, fourcc, fps, (image_width,
image_height))
        rospy.loginfo("VideoWriter initialized. Saving video to:
{}".format(output_file))
```

➔ The __init__ function initializes the video writer that will save frames to a video file. Something that needs to be mentioned is this line: fourcc = cv2.VideoWriter_fourcc(*'XVID'), which sets the codec for video compression (XVID is commonly used for .avi files).

```
    def write(self, frame):
        self.writer.write(frame)
```

➔ The write() function's responsibility is to append a single frame to the video file by calling the self.writer.write(frame), which writes the given frame into the output stream.

```
    def release(self):
        self.writer.release()
        cv2.destroyAllWindows()
```

➔ Lastly, the release() function's purpose is to close the video file and release the system's resources. This works by calling ***self.writer.release()*** which finalizes the video file and

cv2.destroyAllWindows() in order to ensure that any OpenCV windows used during the session will close.

4.10 Main Pipeline

The purpose of the drone_detection_stream.py class is to integrate all the other major classes we have explained so far in order to implement the drone detection, monitoring, estimation and control system in real time.

This class, as will be said later, is connected to the DJI drone's camera in order to receive real-time video stream. In addition to that, it applies object detection using YOLO, tracks drones using Kalman filters, estimates their distance and speed, and sends yaw control commands based on the detected object's position in order to align the camera with the target. Also, for testing purposes the system saves the annotated video and supports PID-based smooth control feedback. In other words, this class acts as an organizer in order to connect all the system's modules and to enable their synchronized operation.

4.10.1 Detailed Breakdown of the functions of the Class

```
    def __init__(self):
        self.IMAGE_WIDTH = rospy.get_param("~image_width", 640)
        self.IMAGE_HEIGHT = rospy.get_param("~image_height", 640)
        self.HORIZONTAL_FOV_DEG = rospy.get_param("~horizontal_fov_deg", 82.9)
        self.focal_length = compute_focal_length(self.IMAGE_WIDTH,
self.HORIZONTAL_FOV_DEG)
        rospy.loginfo("Computed Focal Length: {:.2f}
pixels".format(self.focal_length))

        classes_file = rospy.get_param("~classes_file",
"/home/jetson/Downloads/aiders_osdk/src/dji_sdk/scripts/collaborative_localiza
tion/new_detection/general.names")
        drone_sizes_file = rospy.get_param("~drone_sizes_file",
"/home/jetson/Downloads/aiders_osdk/src/dji_sdk/scripts/collaborative_localiza
tion/drone_sizes.json")
        self.class_names = self.load_class_names(classes_file)
        self.drone_sizes = self.load_drone_sizes(drone_sizes_file)
        missing_sizes = set(self.class_names) - set(self.drone_sizes.keys())
        if missing_sizes:
            rospy.logwarn("Warning: Missing sizes for:
{}".format(missing_sizes))
```

```
        with open("/var/lib/dbus/machine-id", "r") as f:
            boardId = f.read().strip()[0:4]
        self.dronename = 'matrice300_' + boardId

        self.image_handler = DroneImageHandler(self.dronename +
'/main_camera_images', self.IMAGE_WIDTH, self.IMAGE_HEIGHT)
        self.detector = DroneDetector(
            weights=rospy.get_param("~weights",
"/home/jetson/Downloads/aiders_osdk/src/dji_sdk/scripts/collaborative_localiza
tion/new_detection/general_best3.weights"),
            config=rospy.get_param("~config",
"/home/jetson/Downloads/aiders_osdk/src/dji_sdk/scripts/collaborative_localiza
tion/new_detection/general.cfg"),
            netsize=(self.IMAGE_WIDTH, self.IMAGE_HEIGHT),
            conf_thresh=rospy.get_param("~conf_thresh", 0.5),
            nms_thresh=rospy.get_param("~nms_thresh", 0.2),
            classes_file=classes_file,
            gpu=True
        )
        self.tracker = KalmanTracker(iou_threshold=0.5, max_inactive=50)
        self.controller = DroneController(self.dronename)
        output_dir = rospy.get_param("~output_dir",
"src/dji_sdk/scripts/collaborative_localization")
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)
        output_file = os.path.join(output_dir, "drone_detection_stream.avi")
        self.video_recorder = VideoRecorder(output_file, self.IMAGE_WIDTH,
self.IMAGE_HEIGHT)
        self.pid_yaw = PIDController(kp=0.02, ki=0.0, kd=0.005, setpoint=0.0)
        self.last_target_center = None
```

➔ This class illustrates the initialization of all the necessary components of the drone detection and tracking system. To be more specific the class perform the following tasks:

- It loads the system's parameters such as image size, field of view and file paths from the ROS server.
- It computes the camera's focal length dynamically with the help of the appropriate function called compute_focal_length(), which we explained in detail in a previous section of this chapter.

- It loads object class names and physical drone sizes from external files as this information will be needed in a future stage of the system.
- The class also initializes all the helper classes like DroneImageHandler, DroneDetector, KalmanTracker, DroneController, VideoRecorder, PIDController.

```
def load_class_names(self, classes_file):
        if not os.path.exists(classes_file):
            rospy.logerr("Error: Classes file '{}' does not
exist.".format(classes_file))
            sys.exit(1)
        with open(classes_file, 'r') as f:
            names = [line.strip() for line in f.readlines()]
        rospy.loginfo("[DEBUG] Loaded Class Names: {}".format(names))
        return names
```

➔ This function called load_class_names has a very simple purpose as it loads the labels (the drone types) used by the YOLO model by reading each line of the classes_file (a .name file) and stores it as a list. This allows the system to label detected objects by name instead of just numeric class IDs.

```
def load_drone_sizes(self, drone_sizes_file):
        if not os.path.exists(drone_sizes_file):
            rospy.logwarn("Error: Drone sizes file '{}' does not
exist.".format(drone_sizes_file))
            return {}
        try:
            with open(drone_sizes_file, 'r') as f:
                sizes = json.load(f)
            rospy.loginfo("[INFO] Drone sizes loaded successfully!")
        except Exception as e:
            rospy.logwarn("Error loading drone sizes: {}".format(e))
            sizes = {}
        rospy.loginfo("[DEBUG] Drone Sizes Loaded: {}".format(sizes.keys()))
        return sizes
```

➔ Like the previous function, load_drones_sizes have just one simple responsibility. This is to load the real-world sizes (widths) of drone types from a .json file.

➔ This data are crucial for our system as we need them in order to estimate the distance to each detected drone using the bounding box width and focal length.

4.10.2 Main Function of the Class

Last but not least, this class contains the run() function which is the main function that runs in a loop and processes each video frame. In this section there will be a careful summary of the operations that the function orchestrates per frame.

```
frame = self.image_handler.get_frame()
            if frame is None:
                rate.sleep()
                continue
```

➔ In this part of the function, the system get the latest camera frame in order to capture the most recent image from the drone's camera for processing.

```
indices, boxes, class_ids, scores = self.detector.detect(frame)
            detections = []
            for i, box in enumerate(boxes):
                x, y, w, h = box
                x1, y1, x2, y2 = convert_xywh_to_xyxy(x, y, w, h)
                detections.append({
                    'bbox': (x1, y1, x2, y2),
                    'score': scores[i],
                    'class_id': class_ids[i]
                })
```

➔ Here, the system runs object detection using the pre-initialized YOLO model to find drones in the current frame and it returns the indices, the list of bounding boxes, class ids and the scores.

➔ The loop that follows the detection is used to transform each of those detections. By using the convert_xywh_to_xyxy () function it changes the format of the detection for easier tracking and IoU comparison.

➔ Also, it appends the bounding box, confidence score, and class ID into a dictionary, which get stored in a list called detections.

```
active_tracks = self.tracker.update(detections)
frame_out = frame.copy()

display_tracks = [(tid, info) for tid, info in active_tracks if
info['inactive_count'] == 0]
sorted_tracks = sorted(display_tracks, key=lambda t: (t[1]['bbox'][0] +
t[1]['bbox'][2]) / 2.0)
id_mapping = {old_id: new_id for new_id, (old_id, info) in
enumerate(sorted_tracks, start=1)}
```

➔ This part of the function updates the tracker with the latest detections using Kalman filters and it filters out inactive tracks. After that it sorts the visible tracks from left to right based on horizontal position.

➔ Also, it assigns display IDs to each active track.

```
track_control_info = []
for (track_id, info) in display_tracks:
    (x1, y1, x2, y2) = info['bbox']
    cv2.rectangle(frame_out, (int(x1), int(y1)), (int(x2), int(y2)),
info['color'], 1)
    displayed_id = id_mapping.get(track_id, track_id)
    cv2.putText(frame_out, "ID: {}".format(displayed_id),
    (int(x1), int(y1)-10), cv2.FONT_HERSHEY_SIMPLEX, 0.4, info['color'], 1)
    rospy.loginfo("Detection -> Displayed ID: {}, Drone:
{}".format(displayed_id, info['drone_name']))
```

➔ This loop here draws the bounding boxes around each tracked drone and it displays the drone's ID next to the bounding box.

```
if info['drone_name'] == "Unknown":
                    c_id = info['class_id']
                    if 0 <= c_id < len(self.class_names):
                        name = self.class_names[c_id]
                        info['drone_name'] = name
                        info['drone_size'] = self.drone_sizes.get(name, 0.96)
```

➔ Here the system does drone classification and size lookup in order to retrieve from the .json file the drone's type/name and the drone size which will be needed in a later stage for distance calculation.

```
w_box = x2 - x1
distance = (self.focal_length * info['drone_size']) / float(w_box) if w_box >
0 else float('inf')
```

➔ This is where the drone's size is needed as in this part of the code, there is a distance calculation of the drone using the known physical width of the drone and bounding box width in pixels. The calculation formula used here is called the pinhole camera model:

$$\text{Distance} = \frac{\text{Focal Length} \times \text{Drone Real Width}}{\text{Bounding Box Width (pixels)}}$$

```
center_x = 0.5 * (x1 + x2)
center_y = 0.5 * (y1 + y2)
current_time = rospy.get_time()
last_time = info.get('last_update_time', current_time)
```

```
dt = current_time - last_time if current_time - last_time > 0.01 else 0.01

prev_center = info.get('prev_center', (center_x, center_y))
displacement_px = math.sqrt((center_x - prev_center[0])**2 + (center_y -
prev_center[1])**2)
real_disp = (displacement_px * distance) / self.focal_length
speed = real_disp / dt
```

➔ In addition to the distance calculation, we estimate the speed by calculating the drone's center position and by measuring how far the drone has moved in pixels since the last frame. The formulas used for this calculation is the basic kinematic – speed formula and the real displacement pixel-to-metric scaling:

$$\text{Speed} = \frac{\text{Real Displacement}}{\Delta t}$$

$$\text{Real Disp} = \left(\frac{\text{Pixel Displacement} \times \text{Distance}}{\text{Focal Length}} \right)$$

➔ After we make those measurements, we convert pixel movement to real-world movement using distance and focal length and we divide that by time to get speed in meters per second .The focal length formula is shown below:

$$f = \frac{\text{Image Width}}{2 \times \tan(\text{Horizontal FOV}/2)}$$

```
y_text = int(y1) - 10
cv2.putText(frame_out,"Drone:".format(info['drone_name']),(int(x1), y_text-
10),cv2.FONT_HERSHEY_SIMPLEX, 0.4, info['color'], 1)
cv2.putText(frame_out, "Dist: {:.2f}m".format(distance), (int(x1), y_text-20),
cv2.FONT_HERSHEY_SIMPLEX, 0.4, info['color'], 1)
cv2.putText(frame_out, "Speed: {:.2f}m/s".format(speed), (int(x1), y_text-30),
cv2.FONT_HERSHEY_SIMPLEX, 0.4, info['color'], 1)
cv2.putText(frame_out, "Conf: {:.2f}".format(info['score']), (int(x1), y_text-
40),cv2.FONT_HERSHEY_SIMPLEX, 0.4, info['color'], 1)
```

➔ The system also displays textual information in the video frame about each tracked drone and specifically the drone's type, its distance from the camera, its estimated speed and its detection confidence score for testing and evaluation purposes.

```
target_center = compute_group_center(track_control_info)
if target_center is None and self.last_target_center is not None:
    target_center = self.last_target_center
if target_center is not None:
    self.last_target_center = target_center
    image_center = (self.IMAGE_WIDTH / 2.0, self.IMAGE_HEIGHT / 2.0)
    error_x = target_center[0] - image_center[0]
    dt = rospy.get_time() - loop_start_time
    yaw_command = self.pid_yaw.update(error_x, dt)
    self.controller.send_command(forward=0.0, yaw=-yaw_command)
```

➔ The last operation of the system is the yaw correction via PID and control command, in which the system relies on two cases

- If there is just one drone detected, then it will calculate its center distance from the center of the camera's field of view and then it will send rotate commands to the drone in order to have the detected drone in the center of the camera's FOV.
- If multiple drones are being detected, then the system will calculate the average position of all detected drones (the group center) and it will adjust the yaw of the drone to that center in order to have a generic scope of all the drones.

4.11 PC-Based System (Video Processing)

As mentioned in the beginning of the chapter, despite the main system that runs on embedded devices there is also a system that is used for PCs that is processing video to make detections. The idea behind both systems is the same but there are some differences in the PC-Based system:

➔ It uses OpenCV's VideoCapture to read video frames.

➔ It uses SORT (Simple Online Realtime Tracking) instead of Kalman filters.

➔ It includes detailed visualization of drone type, distance, and speed.

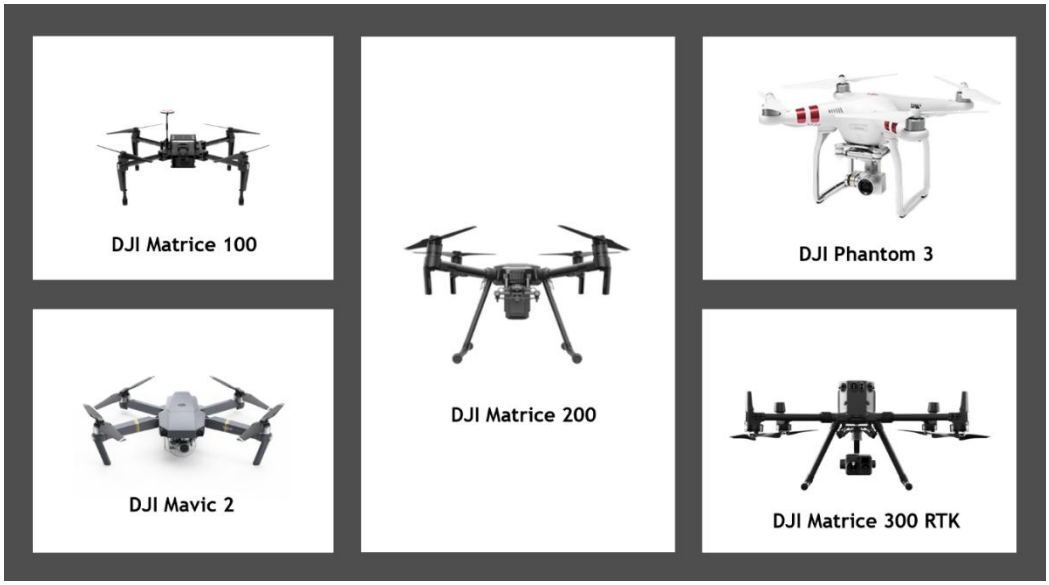
4.12 Training the YOLOv4 Model with Darknet

One of the most important assignments in the development of the drone detection and tracking system was training a drone class-specific object detection model. In this section, the procedure adopted to collect and prepare the dataset, configure the training files, and train the model using the YOLOv4 architecture in the Darknet framework is outlined.

4.12.1 Dataset Collection and Augmentation

The dataset that was used in this project contained approximately 10,000 images. The initial dataset consisted of approximately 7,000 hand-labeled images collected from videos with various drone models at various scenes and from public repositories. Heavy image augmentation techniques were employed to enhance the diversity and balance of the dataset, including random vertical and horizontal flips, change in brightness and contrast, rotation, scaling, cropping, and noise injection. These augmentations simulated varying lighting, drone attitudes, and background, all of which contributed to further enhancing the model's generalization towards detection. To ensure effective model evaluation, the dataset was split into 80% for training, 10% for validation, and 10% for testing, allowing for proper tuning of model parameters and unbiased performance assessment.

All of these tags were provided in YOLO format, which refers to bounding boxes through the use of normalized (class_id, center_x, center_y, width, height) coordinates with respect to image size. The annotation files were auto-generated and were referenced back to corresponding images, facilitating integration with the Darknet training pipeline. The detection model was also trained with 6 various drone classes (5 below and one class is “other-drone”), detailed below according to the definition in the general.names file:



4.12.2 Darknet Training Environment and File Structure

The model was trained using Google Collab Pro, in which I was able to use some high-end GPUs like the NVIDIA A100. The power of those GPUs helped me to finish the training in a few hours instead of a couple of weeks. The training environment used the AlexeyAB fork of the Darknet framework, which supports CUDA, cuDNN, OpenCV, and many advanced YOLO features such as CIoU loss, Mosaic augmentation, and greedy NMS.

The Makefile in Darknet was compiled with the following flags to ensure GPU and OpenCV support: GPU = 1, CUDNN = 1, OPENCV = 1, AVX = 0, OPENMP = 0, LIBSO = 0. This enabled fast training with real-time logging and image augmentation directly in the training loop. In the general.cfg (configuration file) which was based on YOLOv4-tiny we selected the following values for our:

- Image size: 640×640
- classes=6
- filters=33 (computed as (6 + 5) * 3)
- max_batches=12000
- steps=9600,10800
- batch=64, subdivisions=16 (training) and batch=1, subdivisions=1 (inference)
- Additional flags such as mosaic=1, ciou, and greedynms were enabled for better generalization.

Example from the YOLO layer block in the config:

```
[convolutional]
size=1
stride=1
pad=1
filters=33
activation=linear

[yolo]
mask=3,4,5
anchors=5,  5,   7,  8,  10, 10,  13, 12,  22, 19,  73, 64
classes=6
num=6
jitter=0.3
scale_x_y=1.05
cls_normalizer=1.0
iou_normalizer=0.07
iou_loss=ciou
ignore_thresh=0.7
truth_thresh=1
random=1
resize=1.5
nms_kind=greedynms
beta_nms=0.6
mosaic=1
```

➔ This setup ensured that the model was optimized for detecting the six specific drone classes under diverse and realistic conditions.

4.12.3 Model Selection Rationale

YOLOv4 and YOLOv4-tiny were the model architectures chosen to train. These model architectures were chosen after taking the software and hardware constraints in the deployment environment into consideration.

Later implementations of YOLO, i.e., YOLOv5, YOLOv7, and YOLOv8, are mostly implemented in PyTorch and are optimized for execution in top-end desktop GPUs. Though they are more accurate in performance during training, they are computationally intensive and less lightweight in design, and therefore not best suited for use in embedded systems such as the NVIDIA Jetson line. Even specialized variants such as YOLOv5n (nano) are usually based on PyTorch and ONNX runtime and are not easily composable in ROS workflows.

Conversely, YOLOv4, and YOLOv4-tiny, in C/C++ through Darknet, are a more performing solution, better suited for real-time inference in Jetson boards. The latter also come in OpenCV's DNN module, within a dependency- or converter-free environment. The tiny variant, specifically, in fact, performs real-time in even less capable Jetson devices with satisfactory detection of the drones.

Thus, the application of YOLOv4 and YOLOv4-tiny is a design trade-off between accuracy and computationally demanding nature, facilitating simple deployment within low-resource embedded devices in a manner that does not compromise adequate model performance for detecting the drone.

4.13 Summary

Both implementations of the drone detection system, both in embedded platforms and PC platforms, are manifestations of well-designed and modular program architecture. Although designed to run in different environments, the two inherently manifest a common design aspect of aiming to achieve clarity, reusability, and ease of maintenance. Within both frameworks lies a shared YOLO-based detector module and usage of uniform utility methods to deliver functionality such as distance calculation, tracking, and marking of data.

The embedded version is designed with real-time, standalone usage in mind on resource-constrained hardware like NVIDIA Jetson. It is highly integrated with DJI SDK flight control, PID-based yaw compensation, as well as onboard video logging. The embedded version emphasizes low latency, computational performance, and direct hardware access on-board to provide real-world responsiveness.

For contrast, PC-based configuration supports offline or near-real-time processing with less restricted computing resources. It also offers additional functionalities such as advanced visualization, SORT-based tracking, and full-speed/distances measurements on multiple drones. The configuration allows more research and experimentation, as well as performance testing, and thereby is suited to system development or research.

Communication between the modules is handled through the ROS (Robot Operating System) framework, which enables seamless data flow between components using topics, services, and publishers/subscribers. For instance, the image stream from the drone's camera is published to a ROS topic and subscribed to by the detection module, while control commands are published by the PID module to the flight control interface. This modular, message-based architecture ensures loose coupling, real-time synchronization, and the ability to easily extend or replace components without disrupting the entire system.

Every class in the system has a well-delineated purpose, from image recognition and acquisition to tracking, PID control, and ROS communication. Functions are carried out by individual modules, each adding its bit towards a seamless interaction as a pipeline, with a trade-off between speed, accuracy, and extensibility. Not only does this architecture improve development productivity, but also it becomes a simple task to modify or add to the system in the future, by adding new models, sensors, or methods of tracking, for example. Collectively, these two systems provide a robust toolset for real-time drone tracking and localization, accessible for utilization in various operation contexts, ranging from lab experiments to in-the-field deployment.

Chapter 5

Evaluation and Results

5.1 Evaluation Methodology

This section describes briefly the methodology behind the experiments made to measure the performance of the counter-drone system, which consists of two parts:

- ➔ Hardware & software platforms
- ➔ Test scenarios

5.1.1 Hardware & software platforms

For the two implementations of the system the following environments were used:

1. **Embedded platform:**
   - **Device:** NVIDIA Jetson Xavier NX (6-core Carmel ARM CPU @ 1.4 GHz, 384 CUDA cores, 8 GB LPDDR4x RAM)
   - **OS:** Ubuntu-based Jetson Linux
   - **Vision stack:** OpenCV 4.2 (built with CUDA/cuDNN), Python 2.7
   - **Detection model:** YOLOv4-tiny via OpenCV DNN (FP16)
   - **Tracker:** Kalman Filter.
   - **Control SDK:** DJI OSDK 3.10 (via dji_sdk ROS nodes)
2. **Laptop platform:**
   - **Device:** HP Spectre 11th Gen Intel(R) Core(TM) i7-11390H@3.40GHz3.42 GHz
   - **OS:** Windows 11 Home
   - **Vision stack:** OpenCV 4.5, Python 3.11
   - **Detection model:** YOLOv4-tiny via OpenCV DNN (FP32)
   - **Tracker:** SORT (Simple Online Realtime Tracking)

5.1.2 Test Scenarios

1. **Live Drone Flights:** This type of experiment was basically real-time action of the system on a DJI Matrice 300RTK with a Zenmuse H20T camera. It was the most realistic test scenario.
2. **In-Lab testing:** This was the most usual testing method. For this method the system was running on a drone on the ground using also a camera but instead of real-time data it was trying to detect drones from videos on a TV. This method wasn't really helpful for evaluating the system, but it was used for minor tests to see if a new function was working properly.
3. **Recorded Video:** The system was detecting drones from videos instead of real-time video stream. This was extremely helpful when it came to accuracy testing and performance. Also, it was an easy alternative to the first method which wasn't easy.

5.1.3 Experimental Videos

For each experiment , a video was recorded in order to see in live action how the system react to real-time scenarios and what are the results of the detector, the distance and speed results. For testing purposes, three types of experimental videos were taken.

1. **Giving as input to the PC-version system a pre-recorded video:** For the purposes of the thesis, there was the need to make some tests also for multi-detection. For legal purposes it was very difficult to fly 5 drones simultaneously, so a pre-recorded video was used in order to test how the system works with more than one drone. Also, this video has the best resolution as the PC that was used to make those experiments could handle 4K videos, something that the Jetson Xavier couldn't (in the next sections we will see that Jetson's maximum resolution without crashing is 640x640).

Figure 5.1.3.1 about multi-detections

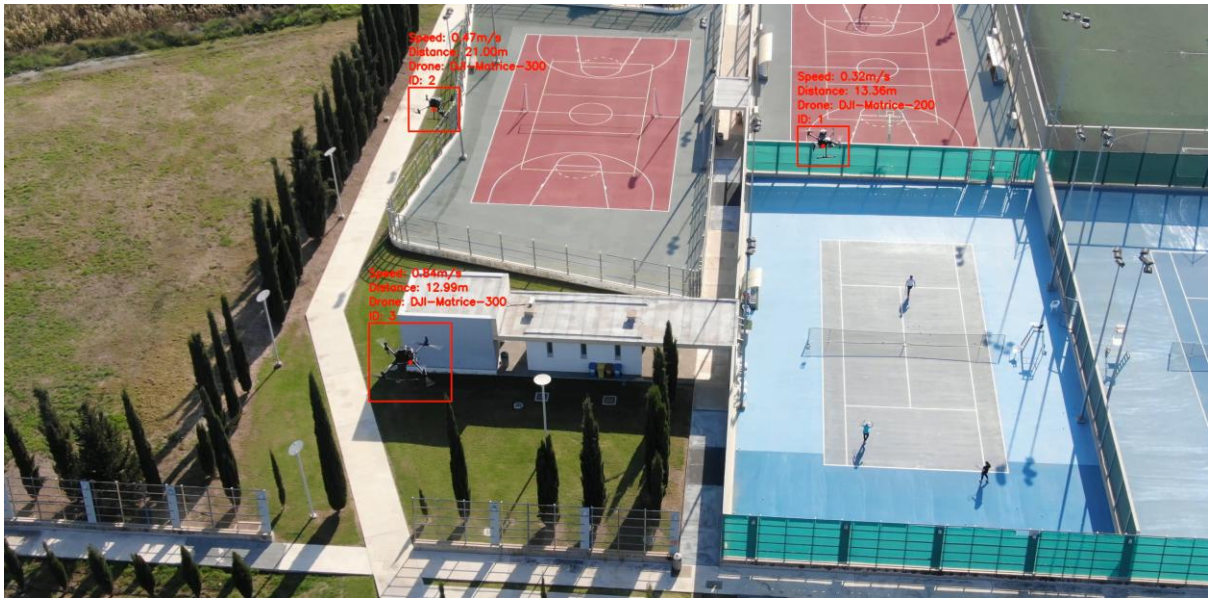


Figure 5.1.3.2 about multi-detections

Both figures 5.1.3.1 and 5.1.3.2 represent screenshots of videos saved during experiments about multi-drone detection. Each drone is successfully identified and annotated with metadata including its unique ID, model type (as we can see that the model is able to identify whether the drone is a DJI Matrice 200 or 300), estimated speed, and calculated distance from the camera. The system demonstrates the ability to detect and track multiple drones simultaneously across different environments, both urban-like settings and open landscapes.

1. **Giving as input to the Jetson-version system a pre-recorded video:** For this experiment, a video of a single drone was used in order to see the success rates of detection in the Jetson version of the system. In order to achieve visible results, lower resolution videos were used (640x640) because the jetson could handle the computational power of higher resolution videos, which resulted in low FPS that made the detections useless.

Figure 5.1.3.3, experiment of videos on jetson

Figure 5.1.3.4, experiment of videos on jetson

In figures 5.1.3.3 and 5.1.3.4 it's clear that the video quality is not the best and sometimes the letters which pinpoints the information of the drone is blurred. As mentioned before, this is due to the lack of computational power the jetson can handle but for that reason there is always log for each frame being processed in order to be able to see clearly the drone's information. Despite the low quality, the detector seems to be able to detect and identify the drone in most cases, which means that the model is well trained.

2. **Real-time flights with the drones:** This scenario is the most realistic as the system is being tested in a real-environment with all the noises aspects out there. Also, this scenario illustrates the PID integration, in which the drone autonomously follows the detected drones by adjusting its yaw (rotation). This experiment is done by connecting the jetson with the drones, so the quality of the image is the same as the second scenario.

Figure 5.1.3.5, real-world experiment with flight

In this experiment, despite the detector, we wanted to test if the PID integration works. After the first detection, the person controlling the drone from its controller loses access as the system take full access of the drone. When the detected drone moves, the system send yaw commands to the drone, which moves accordingly in order to keep its camera aligned with the detected drones at all times.

5.2 Performance Metrics

In order to evaluate the system's performance, there is the need to measure it across four main dimensions: Detection, Throughput & Latency, Tracking , Distance & Speed. For each of these dimensions we will be using some formulas for their evaluation:

1. **Detection:**
   - $\mathbf{Precision} = \frac{\mathbf{TP}}{\mathbf{TP+FP}}$ Which is: Out of all predicted positives, how many were actually correct?
   - $\mathbf{Recall} = \frac{\mathbf{TP}}{\mathbf{TP+FN}}$ Which is: Out of all actual positives, how many did you correctly detect?
   - $\mathbf{F1} = \mathbf{2}\frac{\mathbf{Precission\ x\ Recall}}{\mathbf{Precission\ +\ Recall}}$ Which is a harmonic mean of Precision and Recall, balances both.
   - **Average IoU over Iteration:** Measures how well predicted boxes match ground truth boxes (0 to 1).
   - **Confusion Matrix Count over Iterations:** Tracks TP, FP, FN, and TN over time, gives insight into classification performance.
   - **Mean Average Precision (mAP) over Iterations:** A metric that combines precision across different recall levels for all classes.

2. **Throughput & Latency:**
   - **Inference time (ms/frame):** Average duration of the DNN forward pass (measured with wall-clock timers).
   - $\mathbf{FPS} = \frac{\mathbf{1}}{\mathbf{average\ frame\ processing\ time}}$
3. **Detection , Tracking & Distance calculation:**
   - Realistic experiments were used to evaluate those fields. To be more precise, three methods were used like those mentioned in section 5.1.2.

5.3 Detection Accuracy Results

This section defines the quantitative measurements relevant to the detector's ability to tell apart drones in a heterogeneous set of annotated frames. The training procedure is evaluated by plotting precision, recall, F_1 score, and mean average precision (mAP@0.5) systematically

over different experiments. In addition, numbers of true positives, false positives, and false negatives are carefully tracked for purposes of illustrating the improvement of detection quality as the model continues toward converge. Together, these measurements capture both the system's sensitivity—its ability to well recognize actual drones, along with its specificity, its ability to avoid unnecessary triggers, thus giving an exhaustive evaluation of detection quality over the training procedure.

5.3.1 Precision over Iterations

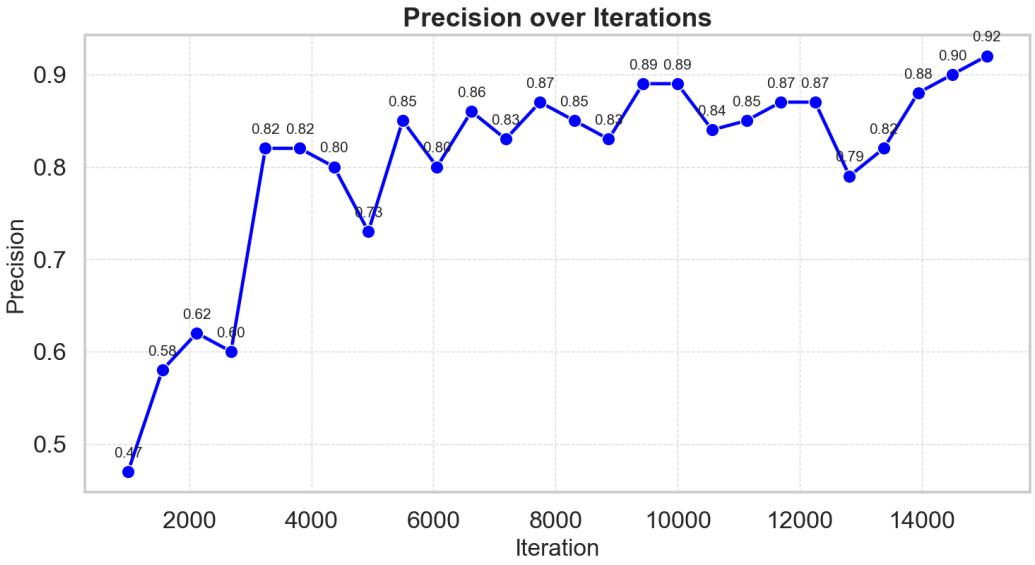


Figure 5.3.1 about Precision

Figure 5.3.1 demonstrates the evolution of precision, i.e. the number of positive detections that were correct. Early in the training process (at around 1,000 – 2,000 iterations), precision is under 0.60 as the network issues many false alarms. Also, in the figure it can be seen that at around 3,000 iterations the model starts to be more precise as the precision jumps to 80%. After 5,000 iterations there are scrambles, ranging from 73% all the way to 89% until the model reaches its peak at 92%. This number appear after 15,000 iterations.

5.3.2 Recall over Iterations

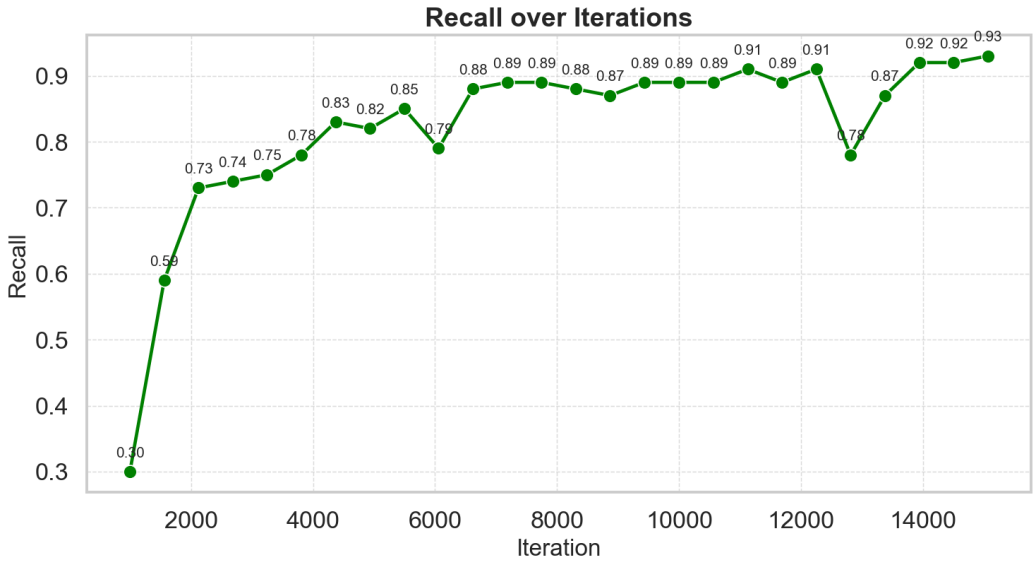


Figure 5.3.2 about Recall

Figure 5.3.2 demonstrates the evolution of recall, i.e. the number of actual positive detections that the system was able to detect. Like precision, recall starts at a low 30% (for the first 1,000 iterations) but after that it climbs rapidly to 73%. By 8,000 iterations recall surpasses 88% and peaks at 93% when it reaches 15,000 iterations, showing that the model ultimately detects over 93% of ground-truth boxes. This steady increase shows that the training dataset and augmentation strategy effectively teach the network to spot small, fast-moving UAVs.

5.3.3 F1 Score over Iterations

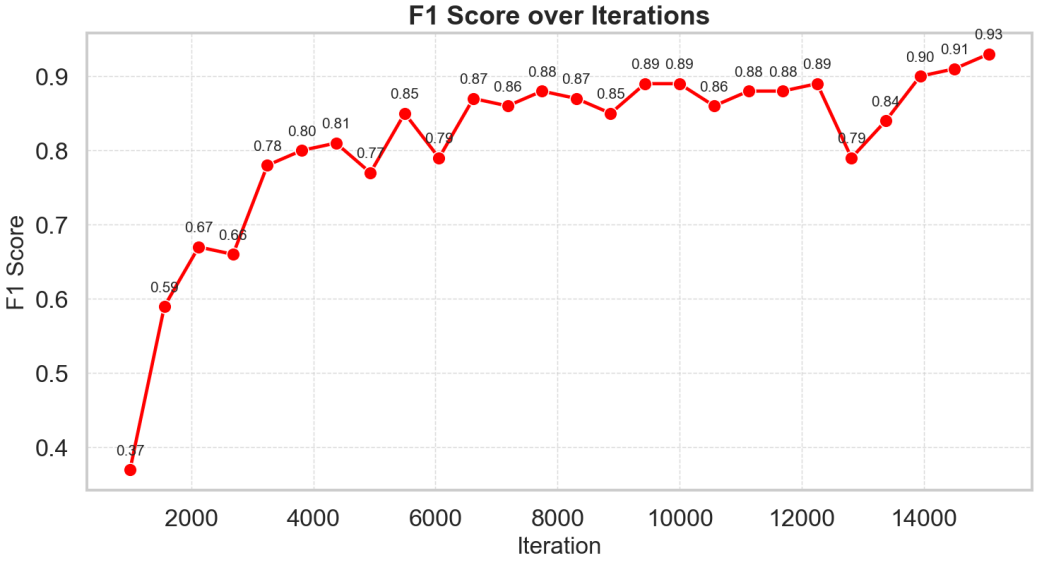


Figure 5.3.3 about F1-Score

Figure 5.3.3 shows the evolution of F1-Score, i.e. the harmonic mean of precision and recall, which begins low again at 37%, like in the previous graphs, and rises over 80% at 4,000 iterations. It can be seen that at 15,000 iterations like in precision and recall f1-score reaches 93%. The F1 curve inherits

both precision and recall trends, confirming that improvements are balanced, false alarms and missed detections both fall over training.

5.3.4 mAP@0.5 over Iterations

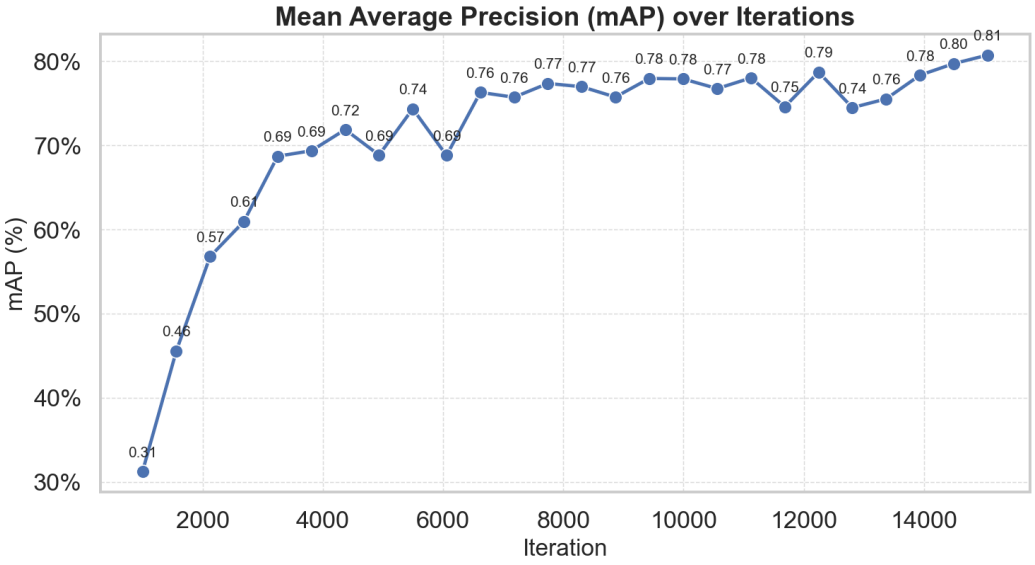


Figure 5.3.4 about mAP

Figure 5.3.4 displays average precision at IoU ≥ 0.5. This is a metric that combines precision across different recall levels for all classes. From 1,000 iterations to 4,000 iterations, we can see mAP increasing exponentially and after the 4,000 mark it becomes steadier until it reaches its peak at 81%. The mAP curve closely tracks f1 above 65%, validating that detection thresholds, NMS settings, and loss functions combine to yield high-quality bounding boxes as well as correct labels.

5.3.5 Confusion Matrix Counts

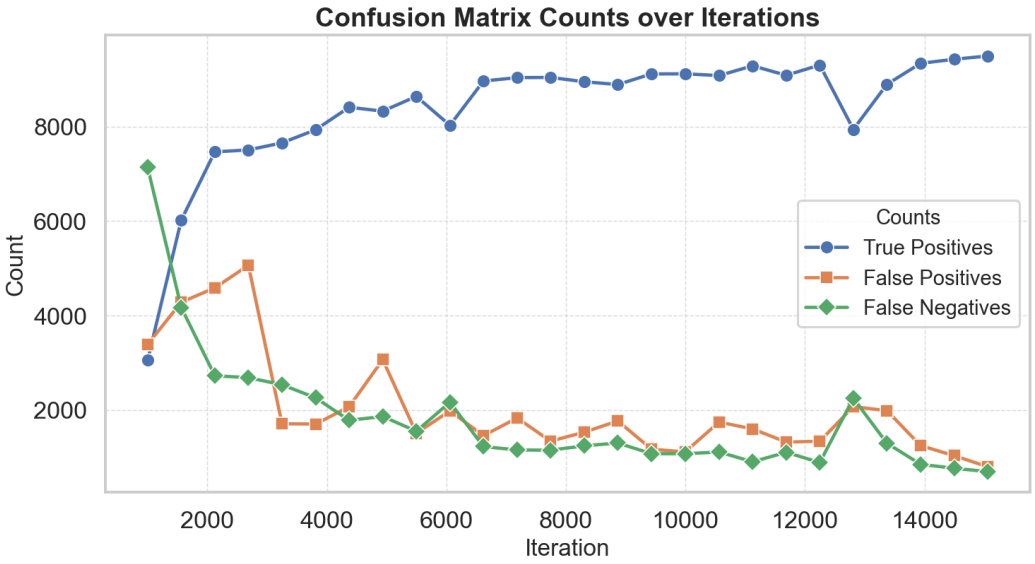


Figure 5.3.5 about Confusion Matrix Counts

Figure 5.3.5 is about Confusion Matrix, which tracks down True Positives (TP), False Positives (FP) and False Negatives (FN) over time and it gives insight into classification performance. The figure shows that for the first 3,000 iterations FP and FN are high, whilst TP are not as high as needed. After 6,000 iterations the plot shows that FP and FN are starting to steadily decline, and TP are steadily increasing. There are also some unexpected behaviors at 6,000 and at 13,000 iterations in which the number of TP drops sharply whilst FP and FN have a sharp peak. This occurs possibly due to bad batches or data issues.

5.4 Real-Time Performance

This section examines the performance of the entire counter-drone system on the integrated NVIDIA Jetson Xavier as well as on the desktop computer. It examines four key aspects: the end-to-end overall speed, the latency at every step, the usage of resources, and the performance change due to varying input resolutions.

5.4.1 End-to-End Throughput

Average frames per second is important to see how much pressure a system handle can. FPS need to be above 5-10 in order for the system to be useful. Below there are some measurements made for both system's versions about FPS. In addition, for the main version, which runs on an NVIDIA Jetson Xavier, tests for multiple resolutions were made in order to see what's the best resolution that the embedded device can handle.

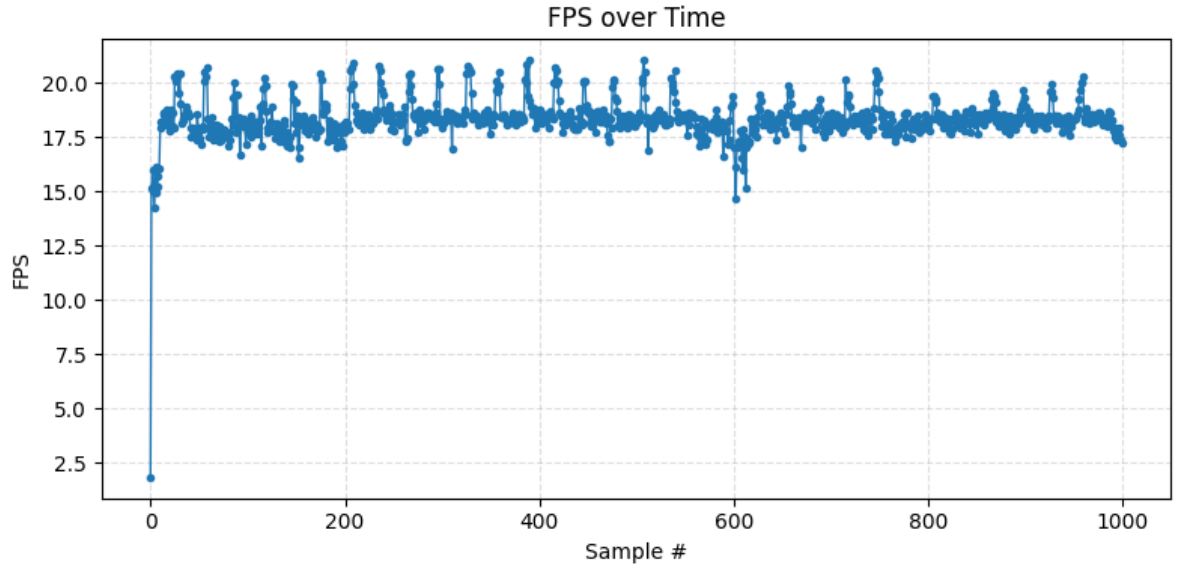


Figure 5.4.1.1 about Frames per second

Figure 5.4.1.1 shows the FPS of the PC-version system. FPS for this system is very good as it almost never fall under 15 fps. In most cases it ranges between 17 – 20 fps. Also, the fact that this system receive 4K videos is worth mentioning, as we can see that it is capable of handling any video.

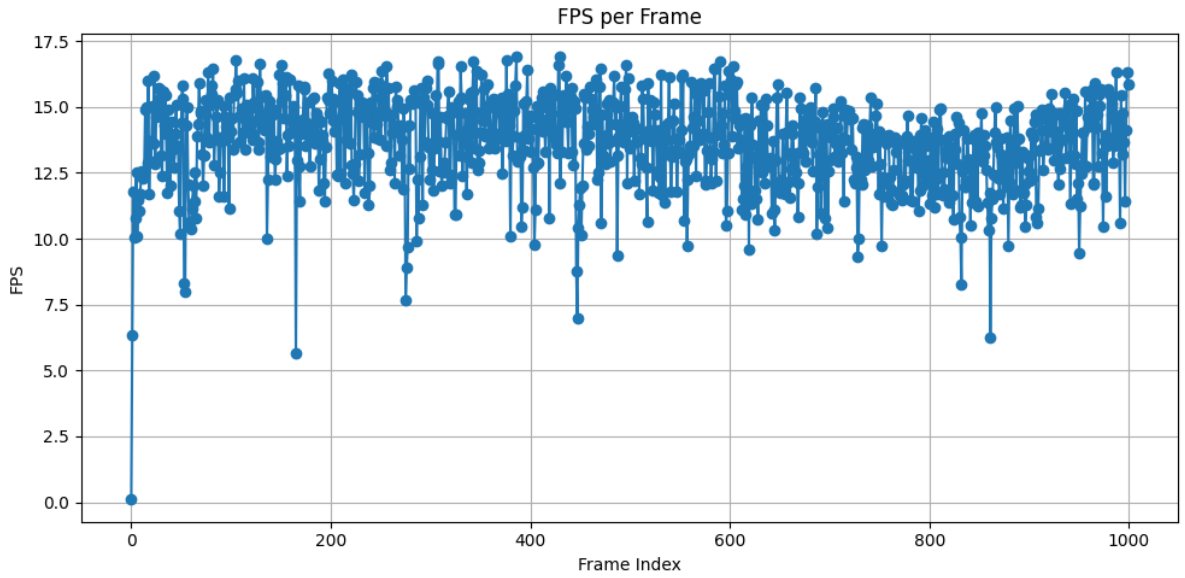


Figure 5.4.1.2 about Frames per second

Figure 5.4.1.2 shows the FPS of the version of the system which run on the Nvidia Jetson Xavier. FPS for this system is more unstable than the previous version. In most cases it ranges between 10 – 15 fps, but on some occasions, it falls under 10 fps. This system, unlike the previous one, receive 640x640 videos. The results are not unexpected as this version is more advanced and integrated with the DJI Matrice 300 RTK. However, it utilizes the ROS CV Bridge for video processing, which contributes to a lower frame rate due to the overhead introduced during image format conversion between ROS and OpenCV.

5.4.2 Latency Breakdown

Profiling at major points of the process: determining the DNN, object updating, the estimation of distance and speed, and screen drawing (which encompasses each cv2.rectangle/putText invocation and resizing the window). Table 5.4.2 provide the average time per frame over 1,000 frames. The following measurements are for the pc-version of the system.

For the PC-version:

Stage	*time (ms/frame)*	*% of total*
Inference	51.7	~70%
Tracking	0.6	<1%
Estimation	0.0	<1%

Drawing	21.0	~29%
Total	73.2	100%

For the Jetson version:

Stage	*time (ms/frame)*	*% of total*
Inference	62.2	~66%
Tracking	0.8	<1%
Estimation	0.5	<1%
Drawing	30.2	~32%
Total	93.6	100%

- ➔ **Inference:** The numbers indicate that the dominant cost is the YOLOv4-tiny forward pass on CPU. According to the calculations, inference on the PC-version is quite faster than the jetson version as this is duo to the hardware of each device such as more powerful CPU/GPU. Despite this, both systems perform well with just a 10 ms difference.
- ➔ **Tracking:** SORT's Kalman and IoU update is essentially negligible. According to the measurements this operation does not contribute to the total latency.
- ➔ **Estimation:** Simple arithmetic for distance and speed adds no appreciable overhead. Like the previous operation, this one also does not contribute to the total latency.
- ➔ **Drawing:** All GUI calls (rectangle, putText, window resize) consume a significant fraction of CPU time. The difference between both systems is significant and contributes to the total latency. Jetson version is slower because more technologies are used in it like mentioned in previous chapters.

5.4.3 Resource Utilization

Aside from the frame rate, one needs to observe how much of the computer's power is consumed by the pipeline. Three readings are made at the same 1,000-frame test at 416 × 416 and 640 x 640 resolution:

- ➔ **CPU load (%):** averaged across all logical cores using psutil.cpu_percent() sampled once per frame.
- ➔ **Memory footprint RAM usage (%):** the process's RSS (resident set size) via psutil.Process().memory_info().rss.

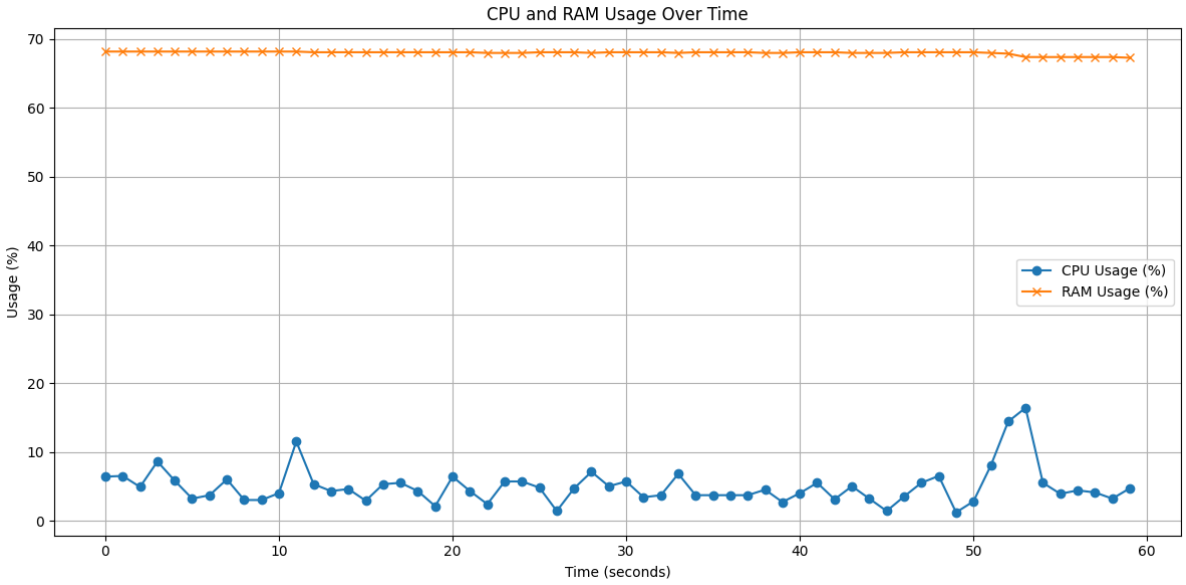


Figure 5.4.3.1 about Resources usage before execution

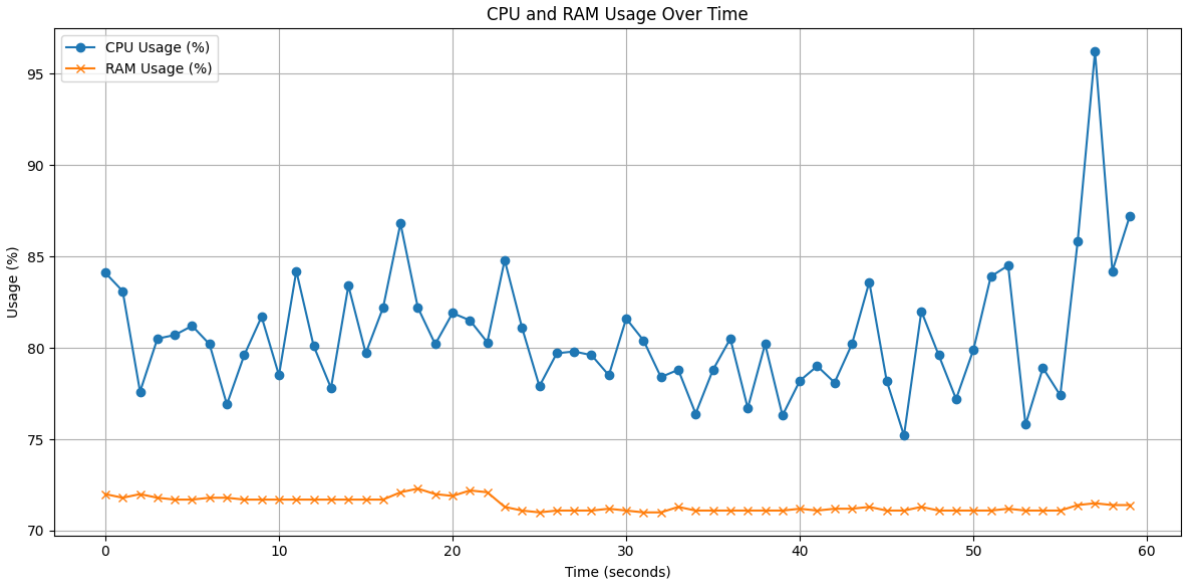


Figure 5.4.3.2 about Resources usage while executing

The figure 5.4.3.1 displays the CPU and RAM usage of the laptop before running the system and the figure 5.4.3.2 shows them while executing the system. It's clear that it takes a lot of computational power in order to run this system as CPU usage increases almost 70%. On the other hand, RAM usage increases by just 5%.

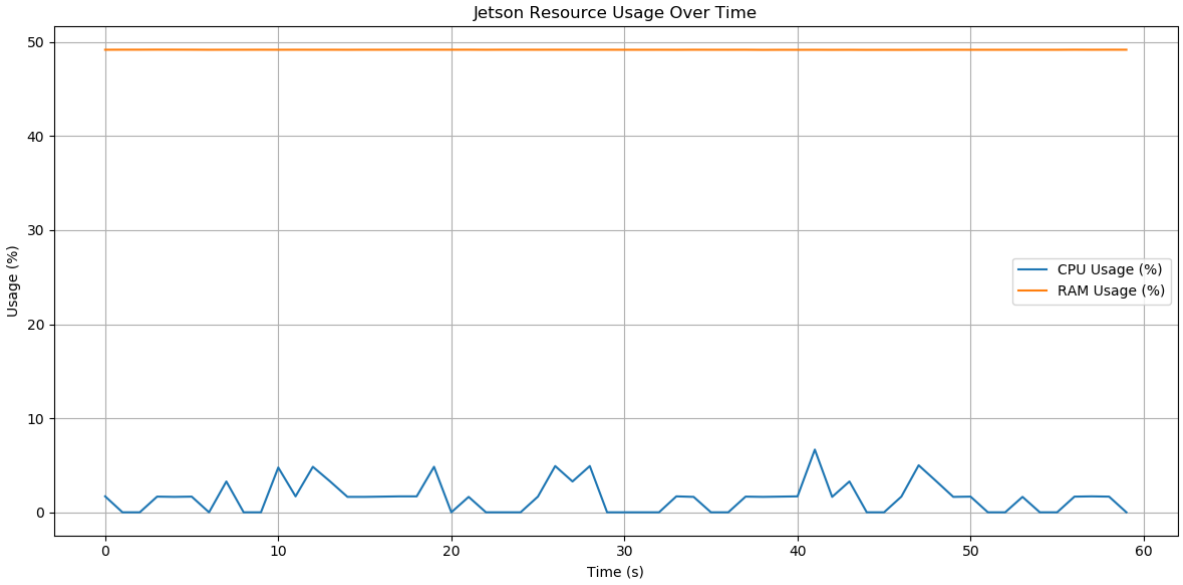


Figure 5.4.3.3 about Resources usage before execution on jetson

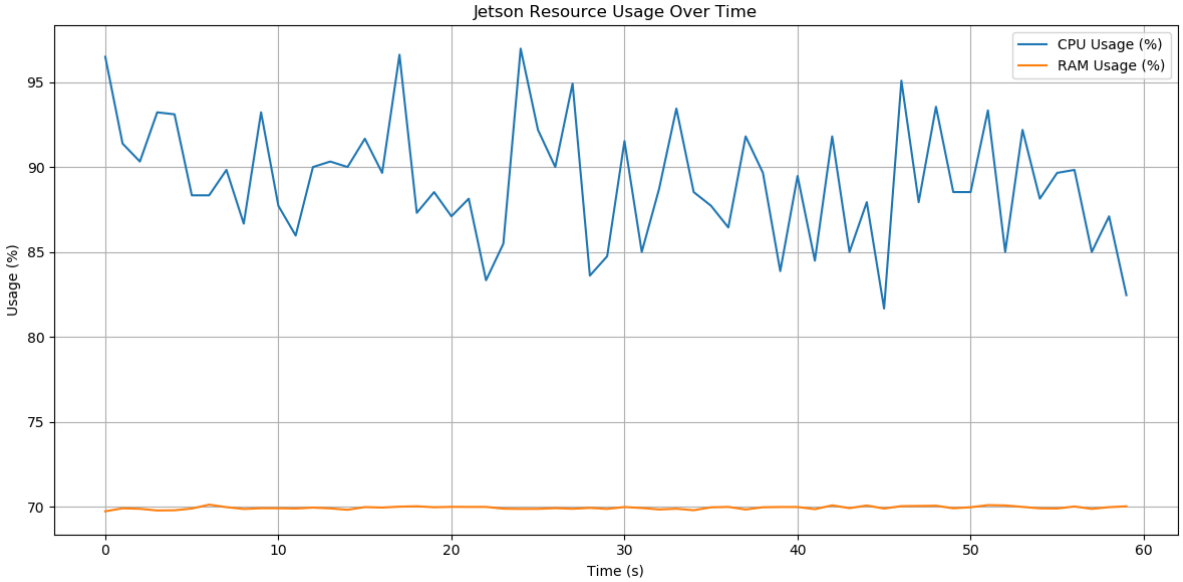


Figure 5.4.3.4 about Resources usage while executing on jetson

These figures illustrate the Jetson device's CPU and RAM usage over a 60-second period before and after the execution of the system. Figure 5.4.3.3 (before execution), CPU usage remains very low, fluctuating between 0% and 6%, while RAM usage stays steady at approximately 49%. In contrast, figure 5.4.3.4 (after execution) shows a significant spike in CPU usage, consistently ranging between 82% and 97%, indicating a high processing load. RAM usage also increases slightly to around 70%, reflecting higher memory demand. These changes suggest that the system places a heavy load on the CPU and a moderate load on RAM.

5.4.4 Resolution Impact

Input resolution has a direct impact on detection accuracy and computation cost. To quantify this trade-off, the procedure was run for 1,000 frames at three typical YOLO input sizes, 416 × 416, and 640 × 640, while recording the average FPS in each case:

416x416:

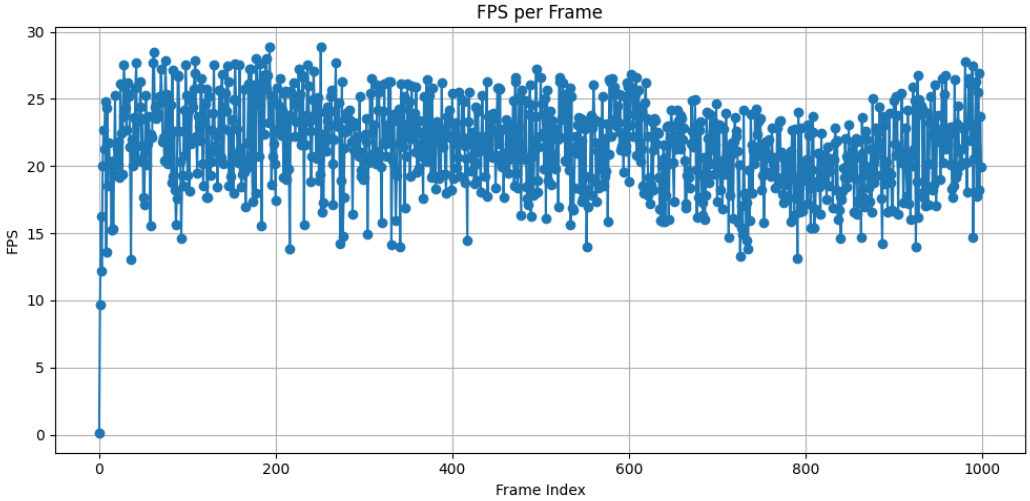


Figure 5.4.4.1 about fps for 416x416 resolution

640x640:

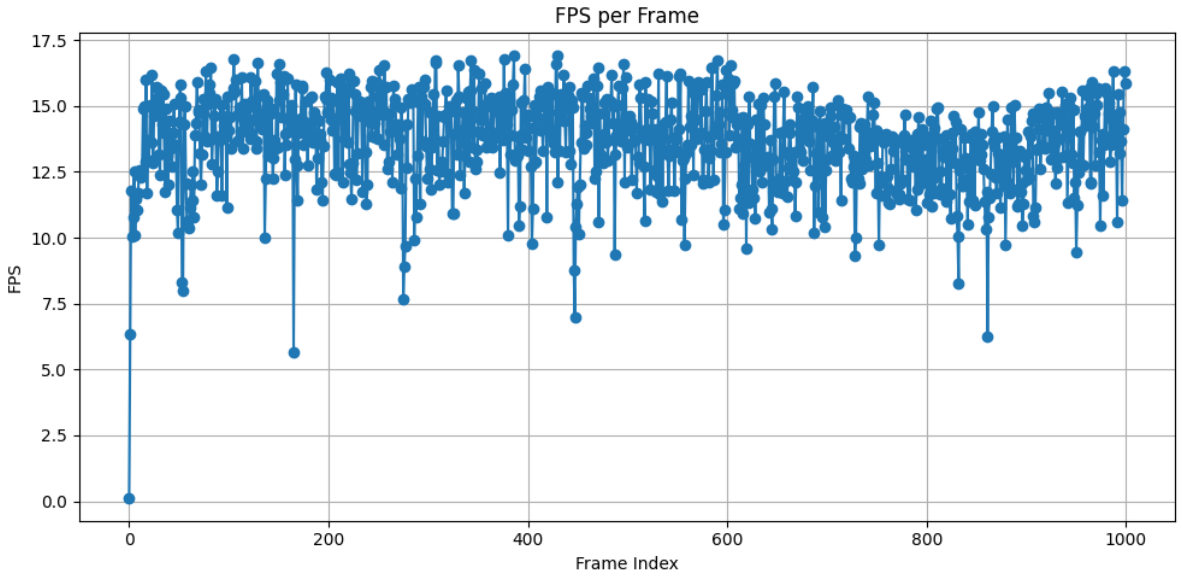


Figure 5.4.4.2 about fps for 640x640 resolution

This trade-off between the system's performance and image in the different resolutions manifests in the disparity between them. The system offers noticeably higher frame rate at the 416×416 resolution, at 20–25 FPS on average with relatively small oscillations. The 640×640 resolution brings the performance down to approximately 12–15 FPS with relatively greater oscillations and dips. This lower performance follows naturally from the higher computational

load of the higher resolution frames. Therefore, the resolution should be a compromise between the accuracy of the detection and the requirements of real-time processing when there might be resource limitations such as in embedded systems.

5.5 Detection , Tracking & Distance evaluation

For multi-drone detection, tracking and distance evaluation three main testing methods were used as already mentioned in previous sections:

1. Offline videos (pre-recorder clips)
2. Lab video stream (indoors in the TV or a monitor)
3. Live flight (outdoor matrices 300 flights for realistic experiments)

Each method had its drawbacks, and that's why all three were used together for the evaluation. First of all, the accuracy of the model with the available dataset is shown in the figure below.

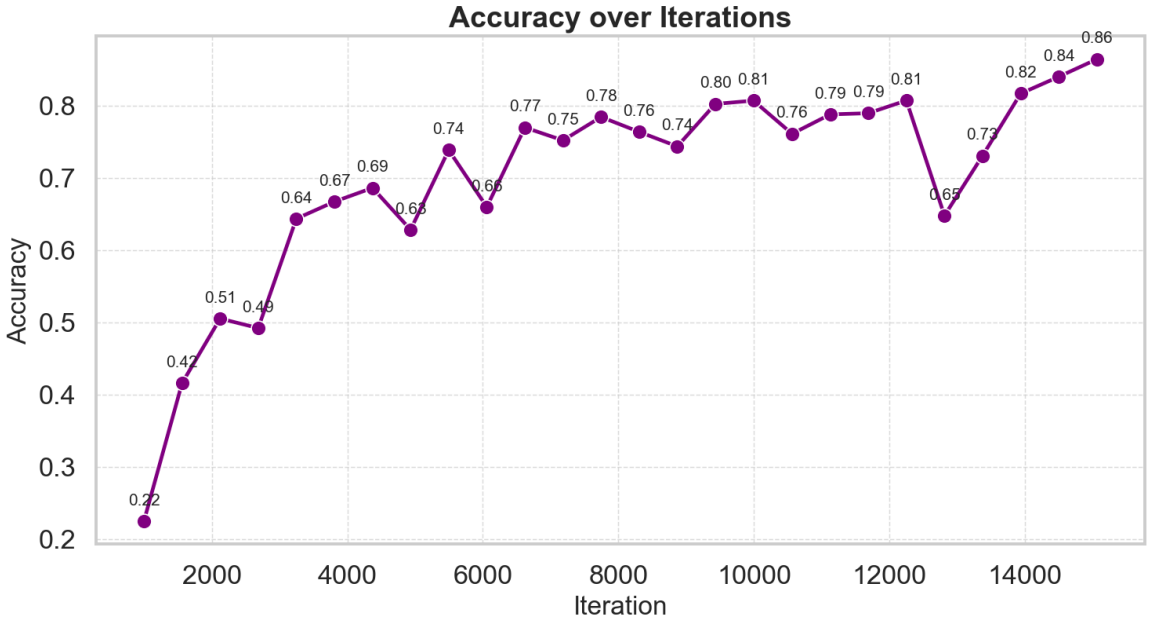


Figure 5.5.1 about model's Accuracy

Figure 5.5.1 indicates the accuracy of the drone detector system across various iterations of the training process. As can be observed, the model begins at relatively poor accuracy levels (about 22%) but gradually improves across iterations to reach an 86% accuracy level at the end. The increasing graph is an indication of learning by the model from the data. Small setbacks like the one observed at about iteration 13,000 can be explained by temporary overfitting or batch data variability. The model quickly recovers, however, and continues learning an indication of health within the training dynamics. The accuracy level reached is reasonable for deployment of the drone detection system to be used under actual applications given constraints on deployment to embedded hardware like the NVIDIA Jetson. The results give confidence to the potential of the model to offer stable and reliable results under actual applications.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

This thesis involves the design, development and enhancement of an integrated drone detection and tracking system for embedded devices like NVIDIA Jetson and for PCs. The aim of the thesis was to improve an already established solution with additional features, fix key weaknesses as well as optimize performance for different development platforms.

One of the significant contributions of this project is the dual-platform architecture:

- The initial deployment runs in real-time on an NVIDIA Jetson on a DJI Matrice 300 drone, featuring detection, tracking, PID-controlled yaw stabilization, as well as video recording of the acquired video.
- The second version operates on an ordinary PC, processing pre-recorded drone footage for multi-drone detection in combination with visual overlays at high-resolution for tracking as well as logging.

Important developments are:

- Integration of YOLOv4 detection module, as trained on custom drone dataset, on Google Collab with Darknet.
- Use of two methods of tracking, Kalman filtering for real-time tracking and SORT for off-line video processing.
- Use of distance calculation of objects based on pinhole camera models as well as kinematic relations.
- Automatic yaw correction PID controller for centering the target found in the middle area of the field of view.
- Modular system design for reuse of code on both platforms as well as for future enhancements.

These improvements greatly enhance the detection robustness, responsiveness, and deployment flexibility of the system, rendering it a more suitable basis for real-world counter-drone systems.

6.2 Key Takeaways

During the system development and testing, the following were observed:

- YOLOv4-tiny has an excellent balance between detection performance and computation efficiency, especially on computation-restricted hardware like the Jetson Xavier.
- Kalman filters assist in achieving smoother, more stable object motion estimates between the frames.
- Under real-time testing, the PID loop, initially restricted for yaw corrections, proved effective in aligning the drone's camera with targets.
- Modular structure and ROS-integration simplifies system debugging as well as real-time communications, especially for robotics applications.

Through the synergy between advanced computer vision algorithms as well as practical control techniques, the system demonstrates an effective avenue for field-deployable drone detection systems.

6.3 Limitations

Despite its advantages, the system also has some limitations, which need to be addressed and possibly corrected in order to achieve optimal system performance:

- Manual assignment of drone size is required in order to calculate distance and speed, which can lead to inaccuracies if a drone is misclassified.
- No altitude or X-Y control: Control of yaw alone is done automatically, restricting complete tracking in 3D space. Also, after the first detection, the drone’s user loses any authority to move the drone. If the user regain authority of the drone, then it will not be possible for the system to control the drone again, only if the system is terminated and restarted again.
- Its performance is degraded in weather or dark environments due to the use of standard RGB cameras.
- Detection performance is dependent on the dataset used for training directly. It will lead to false positives if the dataset is not diverse enough.

- Limited reidentification: the Kalman tracker does not have appearance-based Re-ID, resulting in ID switches in the presence of occlusions. This is difficult to do as the drones we used for testing are identical, as we used mostly DJI Matrice 300s.
- Also, the detection model is limited to 5-6 types of drones as datasets and images are not available online for specific drones and we didn't have many drone types available to add in the dataset.

These boundaries are necessary for future direction in terms of development as well as for efficient deployment in harder environments.

6.4 Challenges Faced

During the implementation of the drone detection and tracking system, several technical and practical challenges were encountered that impacted on the choice of designs as well as the final project result:

- **Hardware limitations in Jetson devices:**

  One major challenge, however, was the computational resource capacity of the NVIDIA Jetson Xavier NX platform. While high for an embedded platform, it could not support full-sized YOLOv4 models in real-time. It is what made us use the YOLOv4-tiny model, whereby detection accuracy had been traded for the sake of stability as well as speed. In addition, Jetson could only display videos of resolution 416x416 and that is no ideal for accuracy. Also, in order to achieve the best accuracy and performance we need to use a newer YOLO version like v8. Newer YOLO versions are less suitable for Jetson Xavier NX due to:

  - Heavier PyTorch-based architecture.
  - Complex conversion to TensorRT/ONNX.
  - High GPU and memory usage.
  - Lower FPS on embedded devices.
  - Poor real-time performance.
  - Integration difficulties with ROS.
  - Incompatibility with OpenCV DNN.

- **Model Training and Dataset Management:**

  It required effort to create a high-quality, balanced dataset. The initial dataset had to be labeled manually, while data augmentation required significant time in order to create higher volumes of varied samples. Maintaining a uniform format of labeling as well as correcting wrongly labeled data were logistical challenges as well. For the thesis I had just two drone types available (DJI Matrice 200 + 300). This made the dataset creation

very hard because I had to find appropriate drone images in flight from the internet, something was very hard as there are not many datasets on the internet that had those type of images. I found a few images that could help (DJI Phantom 3 and DJI Mavic 3) but I managed to collect approximately 6000 images for the dataset and in total 10000 with the image augmentation. In order to achieve the best possible outcome, we need more equipment available to create the appropriate dataset.

- **ROS Integration Challenge:**

  It was difficult to integrate multiple ROS nodes, message types, as well as services, particularly DJI's SDK. Synchronizing timing, image format conversion (with the help of cv_bridge), as well as publishing commands in a reliable manner, were cumbersome to manage. In addition, ROS is only compatible with python 2.7. Because of that, we couldn't use some crucial libraries that could help the system be better like SORT (SORT was only used in the PC-version because of that).

- **Real-world environments for testing:**

  Real-world testing with the drone in actual environmental conditions introduced uncontrollable variables such as light variation, wind, camera vibration, and inadequate space in which to fly in safety. These real-world variations inevitably led to variations in predicted versus actual performance. Also, it was impossible to make real-word test with a flying drone every time that something new was implemented in the system so I the testing method used while developing the system (In lab test using TVs) wasn't so accurate and the distance was very different because of the TVs pixels.

- **PID regulation problems:**

  Control of the yaw axis PID was easy in concept but difficult in reality. It took quite some trial-and-error effort to come up with an acceptable proportional, integral, and derivative term combination, and poor settings initially caused oscillation and divergence.

- **Tracking Edge Cases:**

  At certain instances, the Kalman tracker misattributed detections for close ranges or partially occluding drones. Identity persistence in those instances became difficult without any Re-ID system or appearance features.

- **Inference Integration in OpenCV's DNN Module:**

  Inference with the dNN module of the OpenCV using the YOLOv4 featured model included weight transfer as well as parameter fine-tuning of preprocessing for proper bounded rectangle alignment with real-world objects on the frame. Mild misconfigurations in this respect led to annoying bugs in testing. In spite of such

weaknesses, each issue moved towards the final elucidation of the system components as well as elicited iterative improvement. These issues having been addressed made final deployment robust and transmitted valuable learning experiences that can be transferred in robotics as well as in computer vision endeavors in the future.

6.5 Future Work

While the system is capable of strong real-time detection of drones, there are numerous aspects in which the system can be enhanced and refined so that it is made smarter, proof of scale, and automated as well.

- **Tracking improvement with DeepSORT or ByteTrack:**
  It's a good idea to replace the current Kalman tracker with newer algorithms like DeepSORT or ByteTrack in order to significantly improve object tracking performance. These algorithms include appearance features in addition to motion history, allowing for even more robust ID assignment in the case of occlusions or close proximity interactions. This would significantly reduce ID switches and make multi-drone tracking even more reliable and more accurate.
- **3D Control Integration (Full Motion Control):**
  The current system that has been developed is yaw-based. In future releases, we can implement complete 3D stabilization with the use of altitude and X/Y translational motion can be included. This would allow the drone to track, orbit, or move towards target objects in a dynamic manner, making the system an unmanned interceptor as a whole. If we manage to gain access for every drone movement then we will be able to make our "spy" drone to follow the detected "unwanted' drone.
- **Thermal-RGB Camera Fusion:**
  Combining thermal cameras with RGB camera inputs would increase detection in adverse-visibility environments such as nighttime, fog, or smoke. Object localization can be enhanced as well as false alarms decreased utilizing sensor fusion algorithms in difficult visual conditions. In addition, if we do this, we will be able to make detections at night, which will be very helpful, especially for surveillance purposes.
- **Depth estimation using stereo vision or LiDAR:**
  Using stereo cameras or light LiDAR sensors would allow the system to measure real-world depth in real-time without relying on approximate object sizes. It would eliminate drone-size lookups and significantly enhance distance and speed estimating accuracy.

- **On-Device Model Optimization (TensorRT /Quantization):**
  Using technologies like NVIDIA TensorRT or quantization can reduce the size of the model and speed up inference on devices like Jetson Xavier NX. It can make it possible for big models (e.g., YOLOv4 full or YOLOv5s) to be run in real-time without any performance loss at all.
- **Dataset Expansion:**
  Collect more data in order to achieve optimal results. Also, there is a need to add more drone types in the dataset. If we retrain the model with a much larger dataset, we will make the system more robust in any kind of environment and background. Also, we need to change the embedded device that we use in order to train the model with a newer YOLO version which will be more appropriate for detection systems like YOLOv8.
- **Real-Time Logging and Alerting System:**
  It will be very useful if the system support alerting capability in the cloud (e.g., email, SMS, or MQTT) as it would enable response capability for intrusions in real-time. Logging detection information on drone aircraft type and flight profiles would help in after-the-fact analysis or investigation via forensics.

6.6 Final Thoughts

This thesis is integrating computer vision, robotics, embedded systems and practical problem solving into a single challenge, and its development was a rewarding and unique experience. The development of real-time solutions, such as the drone detection system, was made possible by working with tools such as ROS, OpenCV, Darknet, and DJI SDK.

Apart from the innovations in technology, the thesis further established that existing technologies can be used to address future societal problems, in this case, reduction of the mounting threat of wayward drones. While there is a lot of room for improvement, this system is a viable starting point for future research and applications in the real-world concerning surveillance, border security, emergency response, and other areas.

The experience of learning from the constraints of an existing system and creating a much superior one cemented my technical know-how and also helped me become a competent interdisciplinary practitioner, from robotics to AI and system integration. Excited to see how this can be built upon and transferred to practical applications.

Bibliography

[1] Gertler, J. (2012). *U.S. Unmanned Aerial Systems*. Congressional Research Service. https://sgp.fas.org/crs/natsec/R42136.pdf

[2] Greenwood, W., & Tighe, S. (2018). *Applications of UAVs in Civil Infrastructure*. ResearchGate. https://www.researchgate.net/publication/330601447_Applications_of_UAVs_in_Civil_Infrastructure

[3] Moore, M. (2018). The growing threat of drones: A national security challenge. *Defense & Security Analysis, 34*(4), 390–406. https://scholarlypublications.universiteitleiden.nl/access/item%3A2977691/view

[4] Airsight. *15 reasons to install a drone detection system at your company's infrastructure*. https://www.airsight.com/blog/15-reasons-to-install-a-drone-detection-system-at-your-companys-infrastructure

[5] NQ Defense. *Counter-drone systems for airports: All you need to know*. https://www.nqdefense.com/counter-drone-systems-for-airports-all-you-need-to-know/

[6] Robin Radar Systems. *How to detect drones and mitigate drone incidents at airports*. https://www.robinradar.com/resources/drone-detection-civil-aviation

[7] Robin Radar Systems. *Why traditional radar isn't effective at tracking drones*. https://www.robinradar.com/why-traditional-radar-isnt-effective-at-tracking-drones

[8] Encyclopedia MDPI. *Radar-based drone detection technologies*. https://encyclopedia.pub/entry/53402

[9] Unmanned Systems Technology. *Acoustic drone detection*. https://www.unmannedsystemstechnology.com/expo/acoustic-drone-detection/

[10] Robin Radar Systems. *The pros and cons of radio frequency analyzers in drone detection*. https://www.robinradar.com/blog/radio-frequency-analysers-drone-detection

[11] Zhang, Y., Wang, J., & Li, H. (2025). Machine learning for drone detection from images: A review of state-of-the-art methods. *Neurocomputing*. https://www.sciencedirect.com/science/article/abs/pii/S0925231225004953

[12] Kumar, S., & Singh, R. (2022). Drone audio recognition based on machine learning techniques. *Procedia Computer Science, 213*, 1234–1240. https://www.sciencedirect.com/science/article/pii/S1877050922010225

[13] Medaiyese, O. O., Syed, A., & Lauf, A. P. (2020). Machine learning framework for RF-based drone detection and identification system. *arXiv preprint arXiv:2003.02656*. https://arxiv.org/abs/2003.02656

[14] Howarth, B. (2019). Optical drone detection. *AZoOptics*. https://www.azooptics.com/Article.aspx?ArticleID=1577

[15] D-Fend Solutions. (n.d.). *Anti-drone detection*. https://d-fendsolutions.com/anti-drone-detection/

[16] Seidaliyeva, U., Ilipbayeva, L., Taissariyeva, K., Smailov, N., & Matson, E. T. (2023). Advances and challenges in drone detection and classification techniques: A state-of-the-art review. *Sensors, 24*(1), 125. https://www.mdpi.com/2504-446X/7/8/526

[17] Li, X., Zhang, Y., & Chen, L. (2023). Drone-YOLO: An efficient neural network method for target detection in drone images. *Drones, 7*(8), 526. https://www.mdpi.com/2504-446X/7/8/526

[18] Zhu, L., Xiong, J., Xiong, F., Hu, H., & Jiang, Z. (2023). YOLO-Drone: Airborne real-time detection of dense small objects from high-altitude perspective. *arXiv preprint arXiv:2304.06925*. https://arxiv.org/abs/2304.06925

[19] Diwan, T., Anirudh, G., & Tembhurne, J. V. (2022). Object detection using YOLO: Challenges, architectural successors, datasets and applications. *Multimedia Tools and Applications, 82*, 9243–9275. https://link.springer.com/article/10.1007/s11042-022-13644-y

[20] Liu, J., Plotegher, L., Roura, E., de Souza Junior, C., & He, S. (2024). Real-time detection for small UAVs: Combining YOLO and multi-frame motion analysis. *arXiv preprint arXiv:2411.02582*. https://arxiv.org/abs/2411.02582

[21] Mistry, D., & Banerjee, A. (2017). Comparison of feature detection and matching approaches: SIFT and SURF. *GRD Journals-Global Research and Development Journal for Engineering, 2*(4), 1–6. https://www.researchgate.net/publication/314285930_Comparison_of_Feature_Detection_and_Matching_Approaches_SIFT_and_SUR