Universidad Autónoma de Querétaro
Faculty of Engineering

MAV Autonomous Landing on a Moving Platform Using Deep Learning Algorithms

Thesis

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science in Artificial Intelligence

Presented by:

Alejandro Daniel Matías Pacheco

Supervised by:

Juan Manuel Ramos Arreguin, UAQ, PHD

Co-Supervisor:

José Martínez Carranza, INAOE, PHD

Querétaro, Qro., October 17, 2024

Universidad Autónoma de Querétaro
Faculty of Engineering
Master of Science in Artificial Intelligence

MAV Autonomous Landing on a Moving Platform Using Deep Learning Algorithms

Thesis

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Artificial Intelligence

Presented by:
Alejandro Daniel Matías Pacheco

Supervised by:
Juan Manuel Ramos Arreguin, UAQ, PHD

Co-supervised by:
José Martínez Carranza, INAOE, PHD

Juan Manuel Ramos Arreguin, UAQ, PHD
Chair

José Martínez Carranza, INAOE, PHD
Co-chair

Jesús Carlos Pedraza Ortega, UAQ, PHD
Committee Member

Saúl Tovar Arriaga, UAQ, PHD
Substitute

Efrén Gorrostieta Hurtado, UAQ, PHD
Substitute

Centro Universitario, Querétaro, Qro.
October 2024
México

Dedication

I dedicate this thesis to my family, friends, and loved ones for their unconditional support over the years. Without you, this would not have been possible.

Acknowledgements

I am deeply grateful to my mother, Iraís, for her unconditional support throughout this journey and for teaching me that with dedication and effort, I can achieve anything I set my mind to. To my father, Marcelino, who always believed in me and whose trust allowed me to reach where I am today. To my grandparents, Irene and Aurelio, for being by my side since childhood with their unwavering love, and to my brother Marco, for his constant support and for bringing joy to every family moment. Without you, this achievement would not have been possible.

I thank my best friend, Araceli, who for more than a decade has given me her sincere friendship and unconditional support, despite the distance and specially in the last stage of my master's degree. To my friends at SANC, with whom I have shared my passion for astronomy and scientific outreach, and who have motivated me to continue pursuing my goal of contributing to science.

I am also grateful to the Universidad Autónoma de Querétaro and my professors, whose dedication and teaching provided me with the tools necessary to achieve my goals. In particular, I want to thank my thesis advisor and co-advisor, Dr. Juan Ramos and Dr. José Martínez, as well as my committee members, Dr. Jesús Pedraza, Dr. Saúl Tovar, and Dr. Efrén Gorrostieta, for their time, valuable advice, and feedback. Finally, I thank the Consejo Nacional de Humanidades, Ciencias y Tecnologías (CONAHCyT) for the financial support provided over the past two years, which allowed me to successfully complete this master's degree.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

- **AFAC**: Federal Civil Aviation Agency (Agencia Federal de Aviación Civil)
- **ANN**: Artificial Neural Network
- **CNN**: Convolutional Neural Network
- **COCO**: Microsoft Common Objects in Context
- **CSP**: Cross-Stage Partial
- **C2F**: Cross-Stage Partial with two convolutions
- **DDPG**: Deep Deterministic Policy Gradient
- **FPS**: Frames Per Second
- **GPU**: Graphics Processing Unit
- **GPS**: Global Positioning System
- **INS**: Inertial Navigation System
- **MAV**: Micro Aerial Vehicle
- **MLP**: Multi-Layer Perceptron
- **PID**: Proportional-Integral-Derivative
- **PI**: Proportional-Integral

- **PPO**: Proximal Policy Optimization
- **QR**: Quick Response
- **R-CNN**: Region-based Convolutional Neural Network
- **ReLU**: Rectified Linear Unit
- **ROI**: Region of Interest
- **RPAS**: Remotely Piloted Aircraft System
- **RPN**: Region Proposal Network
- **ROS**: Robot Operating System
- **RGB**: Red, Green, Blue
- **RGB-D**: Red, Green, Blue, Depth
- **SSD**: Single Shot Detector
- **UAV**: Unmanned Aerial Vehicle
- **VOC**: Visual Object Classes (Pascal VOC)
- **YOLO**: You Only Look Once

Resumen

Esta investigación se centra en el desarrollo de un sistema de aterrizaje autónomo para Micro Vehículos Aéreos (MAVs) en una plataforma en movimiento, utilizando técnicas de aprendizaje profundo y visión por computadora. La metodología se basa en conceptos clave y trabajos previos. Un detector basado en You Only Look Once versión 8 (YOLOv8) identifica un código Quick Response (QR) como marcador de aterrizaje, mientras que un Perceptrón Multicapa (MLP) entrenado estima la altura entre el MAV y la plataforma. El sistema integra un controlador Proporcional-Integral (PI), que procesa estas entradas para guiar al dron durante el aterrizaje, utilizando el Sistema Operativo de Robots (ROS) como interfaz. Las pruebas realizadas con un dron Tello demostraron un 90% de éxito en aterrizajes autónomos y una precisión del 97% en la detección del marcador. Estos resultados confirman la efectividad del sistema en entornos dinámicos, demostrando que la combinación de YOLOv8 y MLP proporciona una solución confiable y precisa para el aterrizaje autónomo en plataformas móviles.

Palabras clave: MAV, aterrizaje autónomo, YOLOv8, MLP, marcadores de aterrizaje.

Abstract

This research focuses on the development of an autonomous landing system for Micro Aerial Vehicles (MAVs) on a moving platform, using deep learning and computer vision techniques. The methodology builds on key concepts and prior work. A You Only Look Once version 8 (YOLOv8) based detector identifies a Quick Response (QR) code as the landing marker, while a trained Multi-Layer Perceptron (MLP) estimates the height between the MAV and the platform. The system integrates a Proportional-Integral (PI) controller, which processes these inputs to guide the drone during landing, using the Robot Operating System (ROS) as an interface. Testing with a Tello drone demonstrated a 90% success rate in autonomous landings and a marker detection accuracy of 97%. These results confirm the system's effectiveness in dynamic environments, demonstrating that the combination of YOLOv8 and MLP provides a reliable and precise solution for autonomous landing on moving platforms.

Keywords: MAV, autonomous landing, YOLOv8, MLP, landing markers.

Chapter 1

Introduction

In recent years, Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have experienced a significant increase in various applications, ranging from agriculture and delivery to rescue missions. A subdivision of UAVs, Micro Aerial Vehicles (MAVs), has also emerged as essential equipment for operation in confined spaces or as a low-cost alternative to larger drones.

Simultaneously, with the growing trend of using artificial intelligence to address different problems, various research efforts have been conducted in the area of MAV autonomous landing, aiming to automate the piloting of these vehicles. One of the biggest challenges during the flight of a drone is the landing stage, particularly on a moving platform and in environments where sensors like GPS are unstable or unavailable due to restrictions such as payload capacity.

Precise autonomous landing is vital to ensure the safety and effectiveness of tasks using MAVs. In this context, one of the approaches has been using landing markers along with deep learning computer vision techniques. The basic idea is to detect the landing marker and apply a control algorithm to land on it.

Continuing in this line of research, this study proposes an autonomous landing system based on the YOLOv8 object detector to identify the landing marker, specifically a QR code. The outputs of the detector are then used to determine the position of the drone relative to the marker and to feed a Multi-Layer Perceptron (MLP) to estimate the height, also relative to the marker. With the position in all axes determined, a PI controller is used to land the drone on the QR landing marker.

The system was tested with a Tello MAV, the obtained results demonstrated that it is capable of continuously detecting the landing marker, estimating the height with precision, tracking the moving platform, and landing the MAV on it within an acceptable time. Hence, this system represents a viable solution for autonomous landing in GPS-denied environments or in situations where access for a human pilot is limited or dangerous. The results are promising; however, there is still a wide range for improvement, particularly in the time to land and height estimation.

1.1 Background

Autonomous MAVs landing has been widely addressed with a variety of methodologies, ranging from landing zones detection through a trained model using terrain data sets to landing zone markers search using a trained model to detect them. Also, some works have been conducted to land in moving platforms.

Yu et al. used a Convolutional Neural Network (CNN) based on YOLO and SqueezeNet to infer the coordinates of the center of a landing marker. The CNN was trained on a workstation equipped with a dedicated Graphics Processing Unit (GPU), using images of various landing markers under different lighting conditions. The test hardware was a MAV DJI NAZA F450 with an onboard computer based on an ARM Cortex-A15 processor. The embedded system uses the onboard camera to capture RGB images and feed them to the

CNN. Upon detecting a 50x50 cm landing marker, its coordinates are inferred and converted into angular displacements along the x and y axes, using the camera parameters. These displacements are sent to the PIXHAWK autopilot, and with the help of an onboard sonar (to obtain the MAV's altitude), the x and y positions of the landing marker's center relative to the MAV are calculated. Finally, the autopilot executes the landing routine. The detector model achieved an accuracy of 83.70% with an efficiency of 21 frames per second (FPS), while the landing error between the center of the MAV and the center of the marker was 8.20 cm on the x-axis and 9.11 cm on the y-axis [9].

Lin, Jin and Chen [10] employed a hierarchical approach to detect a 100x100 cm "H"-shaped landing marker under nighttime conditions. In the first stage, the detector uses a Gaussian filter to reduce noise and enhance the image, followed by adaptive thresholding to binarize the image and capture the marker's edges. Subsequently, connected component analysis is performed to select regions of interest (ROIs) in both the enhanced and binarized images. In the second stage, both images are processed through a decision tree that validates the ROIs using four nodes based on shape, pixel ratio, graphic components, and their arrangement. In the third stage, the ROIs are sent to a Convolutional Neural Network (CNN) to confirm if they contain a landing marker. If positive, the vertices' and the "H" center's coordinates are extracted. The detector's performance was evaluated using a DJI M100 UAV connected to a PC with an NVIDIA GeForce 1070 GPU, via the Robot Operating System (ROS) framework, achieving a precision, recall, and F-measure of 97%, 94%, and 96%, respectively. Efficiency was also measured using the same PC and an onboard Nvidia TX2 card, achieving frame rates of up to 40 and 12 FPS, respectively [10].

Nguyen et al. proposed a module composed of the lightDenseNet CNN as a feature extractor and YOLOv2 as a marker detector. YOLO returns bounding boxes that potentially contain landing markers, and the one with the highest confidence score is selected. If the selected bounding box is equal to or smaller than 150 pixels, the coordinates of its center are taken as the center of the marker. If the box is larger, that section of the image is sent to the

Profile Checker v2 algorithm to refine the marker's center location. Profile Checker v2 uses adaptive thresholding and morphological techniques to eliminate noise, and the geometric center is then calculated. Next, a circle is drawn around the predicted center and segmented into sub-profiles based on a threshold, dividing the profile into black and white. Finally, two points called "P" and "Q" are detected, and the marker's direction is estimated using the mid-point "K" of the arc between these points and the detected center. The module's detection performance was evaluated using a DJI Phantom 4 UAV with an onboard camera and a PC equipped with an NVIDIA GeForce 1070 GPU, achieving an average precision and recall of 99%. Additionally, efficiency was measured using the same PC and an onboard Snapdragon 835 GPU kit, achieving frame rates of 40 and 20 FPS, respectively [11].

Cabrera et al. designed a system based on a seven-layer Single Shot Detector (SSD) to detect an "H"-shaped marker and obtain a bounding box whose coordinates are used by the landing controller. The SSD was trained with 5000 RGB images of 320x240 pixels of the landing platform, captured from multiple viewpoints, with rotations, scales, and lighting variations. For training, a PC equipped with an NVIDIA GTX 960M GPU was used. The system was evaluated both externally (using the same PC) and with an onboard computer (Intel Stick M3 without GPU). The tests were conducted with a Bebop 2 MAV, using the ROS framework to establish communication between the MAV and the computer, as well as to calculate and send control signals. In the external PC tests, the system achieved an average detection confidence of 98.43% with an efficiency of 90 FPS, while with the onboard computer, it achieved an average confidence of 98.26% with an efficiency of 13 FPS [12].

Wu et al. approached the autonomous landing on a mobile platform by designing a PID controller aided by deep reinforcement learning (Deep Deterministic Policy Gradient - DDPG) with corrective feedback based on heuristics for fine tuning. Based on previous manual drone landing experience, researchers provided heuristics to the controller, in order to speed up the tuning process performed by the reinforcement learning (RI) algorithm. To estimate the relative position of the MAV to the landing platform, they used a circle in the

landing mark. To detect the landing mark, they developed a system that converts the images captured by the camera to HSV (Hue, Saturation, Value) color model to eliminate all but the blue color, then the HSV mask transform the image to gray scale. Next, a binary image of the landing mark is obtained using threshold segmentation of the gray scale image. In that way, they identified the circle and estimated the coordinates of the center and the diameter, also, combining the diameter and the camera's focal length, they obtained the altitude of the UAV. To validate their approach, they used the Gazebo virtual environment and ROS, the landing platform velocity was set to 0.1 m/s and the MAV velocity to 0.1 m/s and 0.2 m/s, obtaining a landing accuracy of 97% and 93% respectively [13].

Piponides et al. developed a system that relies only on the camera sensor. Their methodology was divided on four stages: object detector, target positioning, simulation and RL based controller. They used the CNN based DroNet detector optimized for UAVs in order to detect the landing mark (a H shaped, red colored helipad) in each frame. After finding the mark, the positional relationship between the landing zone and the MAV is extracted by using the detected bounding box. After finding the center of the bounding box and with some calculations, the rotation angle (a.k.a. horizontal angle) and the vertical angle are obtained, then they ares used to compute the distance with triangulation properties, finally, the speed is calculated. This information is sent to the controller. For the next step, they used ML-Agent RL framework and the Proximal Policy Optimization (PPO) algorithm to train the Neural Network controller in a simulated environment based on Unity game engine, the function of this controller is to output the optimal UAV movement commands. The detector was trained with 3419 real world images captured by a UAV, in different altitudes, angles and lighting conditions, obtaining a 85.9% mean average precision and 95% of accuracy. They tested the system using the DJI Mavie Air UAV and ROS interface to transmit the NN commands from a laptop to the drone. They performed 40 landing intents from different distances from the platform, successfully landing in 30 out of 40 times, reaching an accuracy of 75%. They replicated the test using a NVIDIA Jetson Xavier NX embedded platform instead of the laptop, obtaining similar results [14].

1.2 Problem description

Autonomous landing of Micro Aerial Vehicles is currently an important, growing field of research. With the development of new technologies, such as embedded and more powerful PCs, more accurate sensors and more advanced cameras (stereo, RGB-D), new possibilities arise. Now it is possible to execute highly demanding algorithms on on board flight computers, so the drones be capable of taking decisions in real time.

Currently, most of the drones rely on traditional sensors like GPS, gyroscope, accelerometer and inertial navigation system (INS), however they have constrains that may affect the navigation, i.e. the number of available satellites (GPS) or the loss of accuracy due to the drift error (INS) [15].

One possible solution is the use of multiple sensor to obtain a more robust pose estimation, however, not all those technologies are available for commercial use or they are hard to find in the market, also, MAVs have a very limited payload capacity and it is necessary to reduce to the minimum the sensors on board.

Given the previous constrains, vision systems for navigation have become a focus in research, as visual sensors can bring more detailed information of the surroundings [15] and don´t depend on the availability of external devices. Specifically, monocular cameras are ideal for applications where minimal weight is required.

Regarding the landing process, traditional methods for landing marker detection, pose estimation of the platform and movement controller usually are slow and require previous knowledge of the dynamical model, therefore, approaches using Artificial Intelligence algorithms have gained popularity as they have shown to achieve a high accuracy.

1.3 Justification

Micro Aerial Vehicles (MAVs), commonly referred as drones, have been a widely used tool in military applications for years, going from reconnaissance to fighting tasks. In recent years, with the popularization of MAVs for civilian and research purposes, in addition to the drop of their prices, new fields of applications have emerged.

Civilian tasks for MAVs have been mainly for reconnaissance, search, surveillance and photography, for example, in natural disasters in order to find missing or injured people, as they can flight over hard or non accessible zones or get trough reduced areas. For commercial uses, companies like DHL or Amazon have tested drones for autonomous delivery, using vision systems aided with GPS for telemetry, however, they have found that unfeasible and further research is necessary.

In research field, major efforts have been directed in giving MAVs the capability of autonomous flight, so they can make decisions where pilot aid is not possible. One of the most difficult steps in this process is drone landing (80% of the UAV accidents occur during landing [9]) in unknown environments, where there is no previous information of the terrain or for some reason, use of GPS and other common sensors for navigation is not available or is not enough.

The importance of autonomous landing relies on the advantages that it has in different aspects, socially it could be of help in emergency response situations, i.e. when there is some natural disaster. Autonomous drones can aid in search and rescue operations, taking some first aid medicine or food, potentially saving lives and reducing risks for human. Furthermore, in disasters like floods, the MAV might need to land in some moving surface that is being used as a lifeboat, which can be achieved with the proposed system. In a technical point of view, autonomous landing can be useful for emergency landings in unexpected situations and environmental conditions, or when battery is running low.

One of the main approaches has been the development of vision based landing controllers, ie. optical flow or Machine Learning. In april, 2021, MAVs vision systems for autonomous take off, hovering and landing achieved an important milestone when the Ingenuity mini-helicopter, part of NASA's Perseverance rover for Mars exploration, completed the first autonomous flight in another planet (so far Ingenuity has completed successfully 33 flights), demonstrating the capability and state of the art of this technology.

Therefore, autonomous landing for MAVs is an open field of research for a variety of applications, as many problems involved in existing methodologies need to be addressed, i.e., fast and accurate detection of landing markers or on-board computer limited capacity.

1.4 Hypothesis and Objectives

1.4.1 Hypothesis

The implementation of a deep learning algorithm-based landing marker detector and relative position estimator will result in a significant improvement in the accuracy of detection and pose estimation compared to other systems. This improvement will enable a Micro Aerial Vehicle (MAV) to safely and effectively land on a moving platform.

1.4.2 General Objective

Develop an autonomous landing system for MAVs landing in a mobile platform using Convolutional Neural Networks, landing markers and a monocular camera.

1.4.3 Specific Objectives

- Adapt a existing YOLO architecture to detect landing markers using a generated marker data-set.
- Develop a deep learning based system to estimate the MAV distance relative to the landing platform, trained with data of the on-board monocular camera to control output signals.
- Implement a PI controller using the information extracted by the distance estimator to send the landing commands to the MAV.
- Generate a dataset of synthetic or real data of the MAV's position with respect to the landing marker, with the aid of Gazebo or a Tello drone to train the NN estimator.
- Test the MAV landing system using a Tello drone to verify that it lands safely in the platform.

1.5 Work Organization

The document is structured as follows: Chapter 1 introduced the research problem, providing an overview that included the background, problem description, justification, and the hypothesis, along with the general and specific objectives. Chapter 2 presents the theoretical framework, which covers a review of Micro Aerial Vehicles (MAVs), the relevant details of the software used, and key concepts in deep learning, such as Convolutional Neural Networks (CNNs) and Multilayer Perceptrons (MLPs), concluding with the theory behind Proportional-Integral (PI) controllers. Chapter 3 outlines the methodology used to develop the autonomous landing system, detailing the hardware components, algorithm development, and system integration. Chapter 4 presents the results of the system's performance, evaluating the effectiveness of the landing marker detection, height estimation, and the integrated system. Finally, Chapter 5 offers conclusions drawn from the research and discusses potential directions for future work.

Chapter 2

Theoretical Framework

In order to develop the MAV autonomous landing system, it is required to review the definition of MAV and some of the main characteristics of this type of vehicle, the Official Mexican Standards for the operation of MAVs, the characteristics of Gazebo simulator and the Robot Operating System (ROS) environment, in addition to the concepts of Convolutional Neural Networks. These definitions and topics are approached in the sections below.

2.1 Unmanned Aerial Vehicles

Unmanned aerial vehicles (UAVs) are a category of aircraft designed to operate without a pilot on board. The common UAV includes the airframe, sensor payload and control ground station [16]. UAVs are most used for surveillance, accessing dangerous environments and non-human accessible places or payload delivery. They are classified according to different parameters, like altitude range, size or weight. One of the most popular types of UAVs are the micro aerial vehicles.

2.1.1 Micro Aerial Vehicles

Micro aerial vehicles (MAVs) are defined as a type of aircraft whose dimensions are equal or smaller than 15 cm, or as aircrafts whose flight is characterized by a low Reynolds number

[17]. Due to their size and payload limitations, MAVs are usually controlled with ground control stations, and often they are fitted with a camera, wich makes them ideal for photography, to navigate over disaster areas or to monitor some objective on reduced spaces.

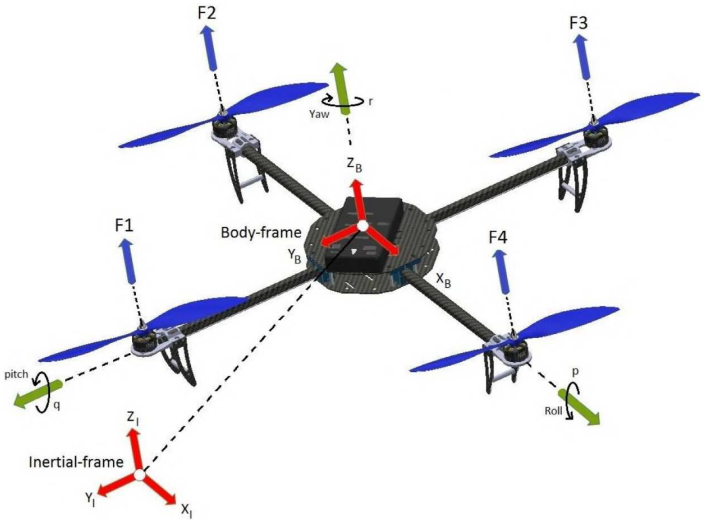

Figure 2.1: Quadrotor MAV coordinate system [1].

MAVs come in a variety of configurations, being the quadrotor one of the best known. Quadrotors consist of four actuators controlled individually to generate relative thrust [18]. The MAV to be used in simulations for this work has six degrees of freedom: X, Y and Z axis translational movements and Pitch, Yaw and Roll rotations around Y, Z and X axis respectively. In Figure 2.1 the coordinate system and degrees of freedom of a typical quadrotor is shown.

2.1.2 Official Mexican Standard for MAVs

In order to operate a MAV safely, the Federal Civil Aviation Agency (AFAC) established in 2019 the NOM-107-SCT3-2019, which establishes the regulations for operating a Remotely Piloted Aircraft System (RPAS) within Mexican airspace. As stated later in Methodology chapter, the tests of the autonomous landing system will be performed via simulation in

Gazebo and flying the MAV in a closed space (defined as a fully enclosed structure that prevents the RPA from accessing the airspace [19]).

According to the NOM-107-SCT3-2019, Section 1, Note 1, this Official Mexican Standard does not apply to RPAS operated in enclosed spaces, and it is the responsibility of the facility owner and event organizer to implement the appropriate safety measures. [19]. Therefore, no special requirements or permissions are needed in order to operate the MAV.

2.2 Robot Operating System

Robot Operating System (ROS) is an open source set of frameworks and libraries used to develop software to program robots, one of its objectives is to facilitate this process by means of modular architecture, software reuse and interoperability between all the elements of the ROS environment [20].

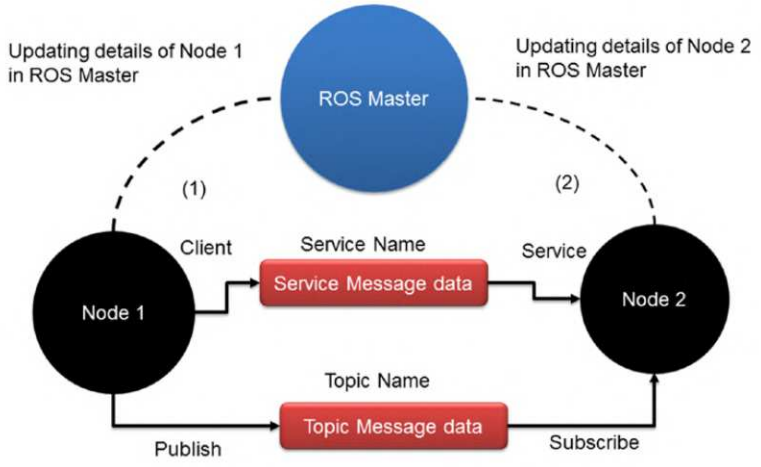

Figure 2.2: Typical ROS communication block diagram [2].

ROS is based on nodes that represent the elements of the robot, they can establish communication by employing the subscribe and publish methods. The nodes are divided in publishers and subscribers, being their functions to send and receive information respectively,

they can send messages (information) through the topics (path or bus) [2]. In Figure 2.2 a typical communication diagram of a system in ROS is shown.

2.3 Gazebo

Gazebo is a visual simulator composed of a collection of open source software libraries used to prototype and test real world applications, design concepts, control strategies, etc. It provides realistic scenarios in conjunction with a dynamics and kinematics physics engine.

A set of plugins is provided by Gazebo in order to integrate it with ROS, supporting various existing sensors. The plugins are compatible with ROS, therefore, the nodes written within it are compatible with simulation, logged data and hardware [20]. In Figure 2.3 a model of the Parrot Beebop MAV in Gazebo environment is presented. An important feature of Gazebo is that the applications tested on it are easily deployed in physical models with minimal changes.

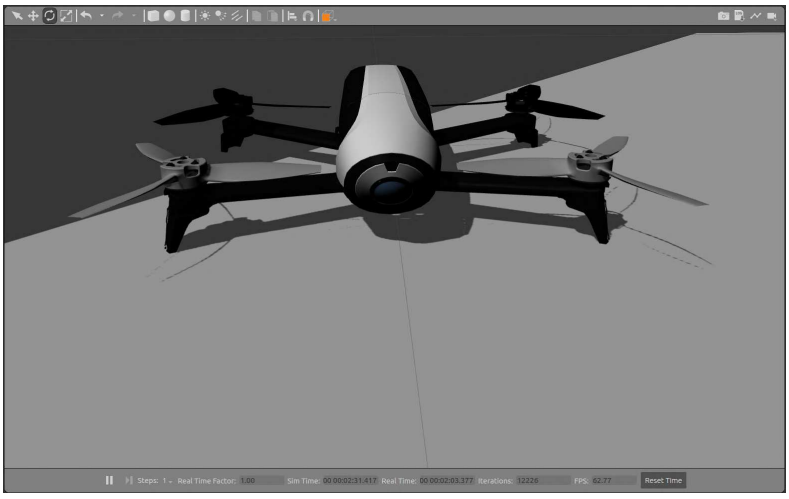

Figure 2.3: Parrot Beebop MAV in Gazebo virtual environment.

2.4 Deep Learning

Machine learning involves techniques for identifying patterns from large datasets and making predictions on new data based on the knowledge acquired from analyzing previously available data. Usually the process begins when the raw data is processed to extract relevant information. According to the learning type, it can be classified into three categories:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning

Traditional machine learning architectures, also referred as shallow learning, are commonly limited by the feature extraction stage, as it is necessary to manually build an algorithm to obtain the most relevant information, transforming the raw data into a suitable feature vector from which the learning system can detect or classify patterns. Usually, an expert in machine learning and an expert in the research field is necessary.

In contrast, deep learning systems require little or null intervention in order to process raw data and extract features. They are capable of learning different features across several layers of abstraction through the hidden layers of the architecture. Deep learning algorithms seek to discover effective representations of the data, where higher-level features are expressed in terms of lower-level features.

The lower layers of deep learning models capture the foundational representation of the data, while the higher levels are constructed on the previous layers and contain more complex or abstract features. That is, the architecture is hierarchical. After extracting the features automatically, the output layer categorizes the data and generates an output label [3]. The variations between conventional machine learning and deep learning architectures are summarized in Figure 2.4.

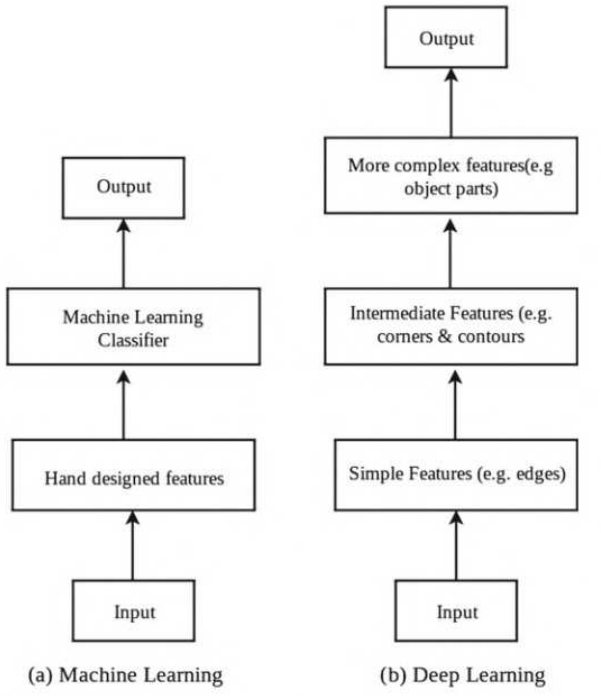

Figure 2.4: Comparison between common machine learning and deep learning architectures [3].

2.4.1 Artificial Neural Networks (ANN).

ANNs are a collection of computational systems modeled after the learning mechanisms and structure of biological neural networks. These systems are particularly adept at learning patterns from data, generalizing from experience, and making predictions without the need for explicit mathematical models of the underlying data. They are also capable of processing degraded or incomplete data [21]. There is a wide variety of neural network models. One of the most popular models is the Multi-Layer Perceptron (MLP), a type of feedforward neural network.

2.4.2 Multilayer Perceptron (MLP)

The MLP is composed of the following characteristics: An input layer or feature vector x, a finite number of hidden layers, an output layer, and a set of weights and biases between each layer, w and θ respectively, and an activation function for each hidden layer [22]. A basic representation of a MLP Neural Networks is presented in Figure 2.5.

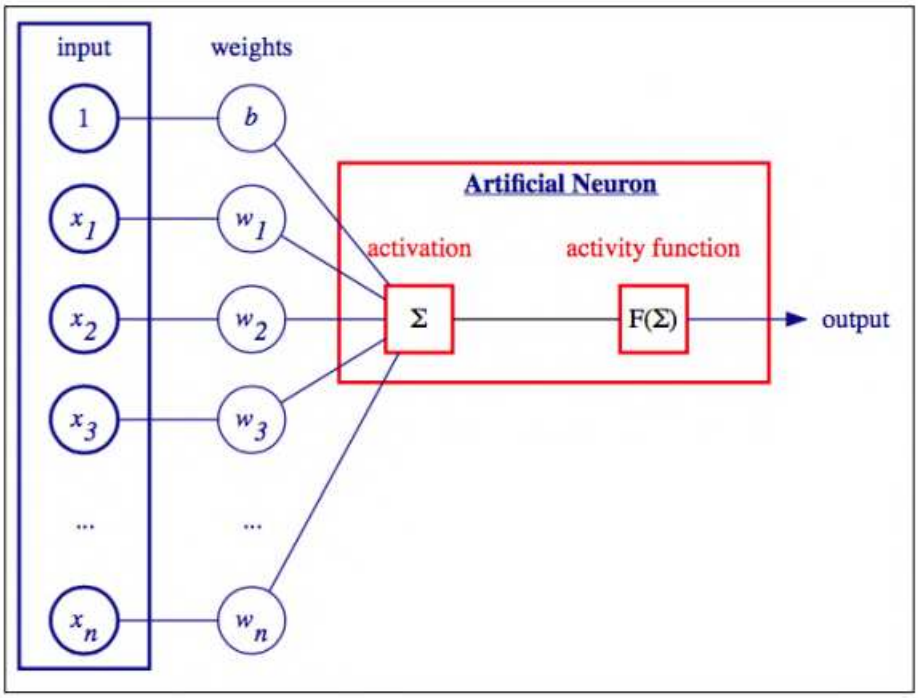

Figure 2.5: Basic structure of a MLP Neural Network [4].

The most basic activation value for a neuron is defined by $a(x) = \sum_i w_i x_i$, where x_i is the value of every input neuron, and w_i is the strength of the connection between neuron i and the output. If the bias is to be included and presented explicitly, the previous equation is rewritten as $a(x) = \sum_i w_i x_i + b$. The bias shifts the decision boundary (or hyperplane) defined by the weights, allowing the model to better fit the data by ensuring the boundary doesn't need to pass through the origin. The activation value can be understood as the neuron's internal state.

The activation value defined earlier can be viewed as the dot product of vector w and vector x. A vector x is orthogonal to the weight vector w if $< w, x >= 0$, and thus, all vectors x satisfying $< w, x >= 0$ form a hyperplane in $\mathbf{R}^n$ (where n is the dimension of x).

Hence, any vector x that satisfies $< w, x >> 0$ lies on one side of the hyperplane determined by w. In this way, a neuron acts as a linear classifier, activating when the input exceeds a specific threshold or, from a geometric perspective, when the input lies on one side of the hyperplane defined by the weight vector.

A neural network can consist of an unlimited number of neurons, but in traditional networks, all neurons are organized into layers, regardless of their quantity. The input layer represents the dataset or initial conditions. Each input neuron is connected to every output neuron by a line, where the value is modulated by the artificial synaptic gap, represented by the weight $w_{i,j}$, which links input neuron x_i to output neuron y_j. Generally, each output neuron corresponds to a class. It's important to note that neurons within the same layer are never interconnected; rather, they are all connected to the neurons in the subsequent layer, and this continues through the network.

2.4.3 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are perhaps the most widely used deep learning system, originally designed for image analysis. They are feed-forward artificial neural networks with a hierarchical structure, which uses convolution operation to learn feature representation. A CNN consists of three types of layers: convolutional layers, pooling or subsampling layers and a fully connected layer. In Figure 2.6, the general architecture of a CNN is presented.

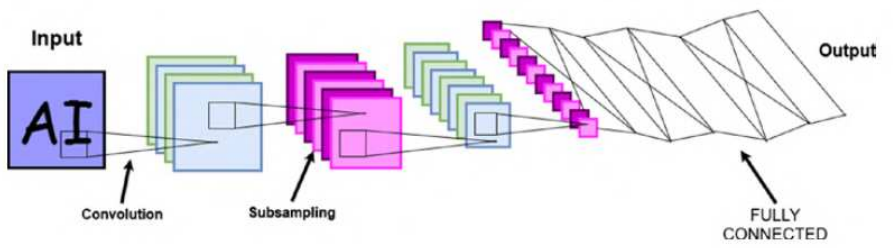

Figure 2.6: Main layers of a Convolutional Neural Network [5].

Convolutional Layer

The convolutional layers are the most important section and they are in essence, a collection of filters, also referred to as kernels. The purpose of convolution is the extraction of features of the input images by mixing them with the kernels. The kernel is a grid of discrete values, it is moved over the whole image laterally and vertically, performing the dot point between them, in order to obtain a single scalar value.

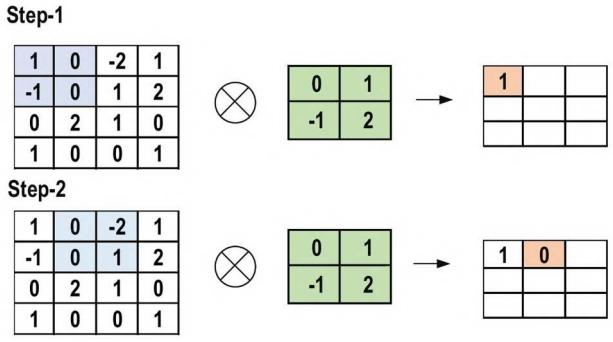

Figure 2.7: First steps of the steps in the convolution process, the image values are multiplied by the kernel and then added to obtain a scalar [6].

In Figure 2.7 the first two steps of the convolution are illustrated, the stride value in this case is one, but another can be set. Due to the convolution layer, rotational invariance, translation invariance and scale invariance is obtained [5].

Pooling Layer

The goal of the pooling layer is to sub-sample or shrink the feature maps generated after the convolution stage. This helps the CNN to determine if certain feature is in an input image, most of the dominant features are kept, although as the layer focuses on finding the location of the feature, relevant information may be missed.

There are many methods of pooling to be used in the pooling layers, however, the most popular are the max pooling, average pooling, min pooling and Global Average Pooling (GAP). In figure 2.8 some of this operations are presented.

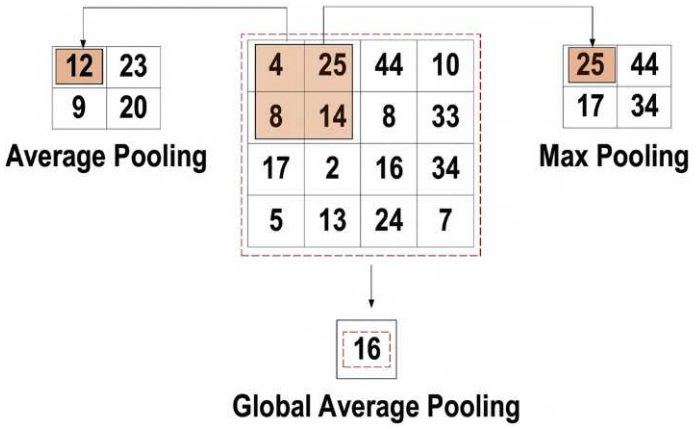

Figure 2.8: Average, GAP and Max pooling methods [6].

Fully Connected Layer

The fully connected layer (Figure 2.9) is a feed-forward neural network situated at the end of the CNN architecture, where each neuron is connected to all neurons from the preceding layers. It receives as input a vector containing the flattened features identified and extracted previously by the convolutional and pooling layers. To produce the class prediction, it uses a nonlinear activation function [5]. The output of this layer contains the class probabilities prediction, thus, the output of the CNN [6].

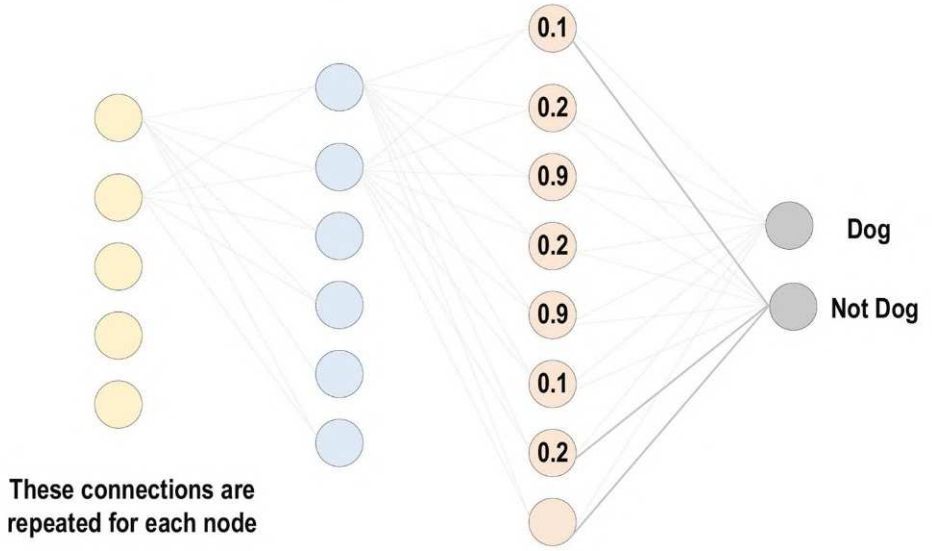

Figure 2.9: Fully connected layer of the CNN [6].

Activation Function

The activation function's role is to map the neuron's input to its output, this output can be interpreted as the probability of the neuron to be activated given a specific input. The input is calculated by performing a weighted sum of the neuron's input and adding the bias term (if it is used) [6].

In a CNN architecture, nonlinear activation layers are used after the convolutional and fully connected layers, also known as learnable layers. Some of the most used activation functions in CNNs are the following:

- Sigmoid: It receives real numbers as inputs, retrieving as output a value between zero and one. It is described by Eq. 2.1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

- Tanh: Similar to sigmoid, it receives real numbers as inputs, but retieves a value between -1 and 1 as output. It is described by Eq. 2.2.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

- ReLU: It converts the input values to positive numbers, acting as a threshold at zero. It is the most used function in CNNs, and its main benefit is the lower computational load. It is described by Eq. 2.3.

$$f(x) = max(0, x) \tag{2.3}$$

Loss Function

The loss function is utilized to compare the difference between the predicted output of the CNN model and the actual output, with the goal of minimizing this loss during the training process. The loss function accepts two parameters as input, the estimated output (prediction) and the actual output (label). There are various types of loss functions, some of them are the following:

- Cross-Entropy, Soft Max or Log loss function: It is commonly used to measure the performance of CNN models. Its output is the probability p, with values between 0 and 1. It can also be used as substitute of the square error in multi-class classification. When used in the output layer, the softmax activations are used to produce the output as a probability distribution [6]. Eq. 2.4 describes the output class probability:

$$p_i = \frac{e^{a_i}}{\Sigma_{k=1}^{N} e_k^{a_i}} \tag{2.4}$$

  where e^{a_i} is the unnormalized output from the previous layer and N is the number of neurons in the output layer. Thus, the cross-entropy is represented by Eq. 2.5.

$$H(p, y) = \Sigma_i \, y_i \, log(p_i), \quad i \, \epsilon \, [1, N] \tag{2.5}$$

- Euclidean loss function: Also known as mean square error, it is commonly used in regression problems. It is defined by Eq. 2.6.

$$H(p, y) = \frac{1}{2N} \Sigma_{i=1}^{N} (p_i - y_i)^2 \tag{2.6}$$

Hyperparameters

Hyperparameters are the externally configurable variables that define the architecture and behavior of a deep learning model but are not directly learned from the dataset. These hyperparameters are important for controlling the capacity and efficiency of the model during training and inference [23]. Regarding MLPs and CNNs, some common hyperparameters include:

- Number of layers and layer size: Determines the depth and width of the neural network, respectively. In MLPs, this refers to the number of hidden layers and the number of neurons in each layer. In CNNs, it refers to the number of convolutional layers, pooling layers, and fully connected layers.

- Learning rate: Defines the magnitude of adjustments made to the network weights at each training step. An appropriate learning rate is crucial to guarantee the network converges efficiently towards an optimal solution without excessive oscillations.

- Batch size: Specifies the number of training examples propagated through the network before a weight update is performed. It influences training stability and computational efficiency.

- Activation function: Determines how the output of each neuron in the network is calculated. Common examples include ReLU (Rectified Linear Unit) for hidden layers and softmax for the output layer in classification tasks.

- Regularization: Includes methods such as L1 and L2 regularization, which penalize large weights to reduce the risk of overfitting.

- Filter size and stride (CNNs): In CNNs, filter size and stride determine the size of extracted features and the spatial resolution of each convolutional layer's output.
- Padding: Specifies how to handle the edges of input images in CNNs to ensure that convolutional layers maintain the desired spatial size.

The election of model architecture and hyperparameters significantly impacts the performance across diverse applications. Careful selection of activation functions is important as they determine the network's capacity to capture complex patterns and gradients. The choice of pooling and convolutional layers determines the capacity to extract hierarchical features from input data. Moreover, the incorporation of fully connected layers needs to be done carefully, as they play a critical role in connecting the extracted features to the final output [6].

2.4.4 Object detectors

Object detection is the process of precise locating and identifying visuals in a picture [24] using bounding boxes. It has a direct relation with object classification, semantic segmentation and instance segmentation [25]. There are two main approaches in object detectors:

- Two-stage detectors: They use a Region Proposal Network (RPN) in the initial stage to generate regions of interest. In the subsequent stage, these regions are categorized, and bounding box regression is performed [26]. Models in this category include Region-based Convolutional Neural Networks (R-CNN), Mask R-CNN, and Faster R-CNN, among others. These models achieve higher accuracy rates but are usually slower [24].
- Single-stage detectors: They address object detection as a regression problem. These models take an input image and directly generate region proposals. During the generation of these regions, the model simultaneously learns the class probabilities and the bounding box coordinates [24]. The main exponents of this category are Single Shot Detector (SSD) and You Only Look Once (YOLO) variants. They are faster than two-stage detectors, but achieve lower accuracy rates.

2.4.5 You Only Look Once (YOLO) v8

YOLO was proposed in 2015 by J. Redmon et al [25]. This algorithm divides the input image into a grid and, in a single evaluation, predicts bounding boxes and class probabilities for each cell [26]. YOLO was originally trained on the ImageNet dataset for classification and then optimized on the Pascal VOC dataset for detection tasks [27].

YOLO v1 splits the image into an SxS grid. Each grid cell predicts C class probabilities, B bounding boxes, and confidence scores for those boxes. The output tensor has the shape SxSx(5B + C), where each bounding box prediction includes 5 elements: 4 coordinates and 1 confidence score. The architecture of YOLO v1 consists of 24 convolutional layers followed by 2 fully connected layers. The network accepts an input image of size 448x448 pixels [25].

YOLO has been in constant development, and since its presentation, different versions have been proposed. In this research, YOLOv8 is used. This variant was introduced by Ultralytics in 2023, keeping a backbone architecture similar to prior versions, but introducing enhancements in the Cross-Stage Partial (CSP) Bottleneck with 2 convolutions module (also called C2 module), now renamed as C2f module. This modified module integrates high-level features with contextual information, thereby improving detection accuracy, focusing on improving execution speed without sacrificing performance. The architecture of YOLOv8 is presented in Figure 2.10.

YOLO v8 also includes a semantic segmentation module named YOLOv8-seg, which consist of CSPDarknet53 feature extractor followed by a C2F module. In the MS COCO dataset test-dev 2017 benchmark, YOLOv8 achieved an average precision of 53.9%, with a processing speed of 280 FPS [27].

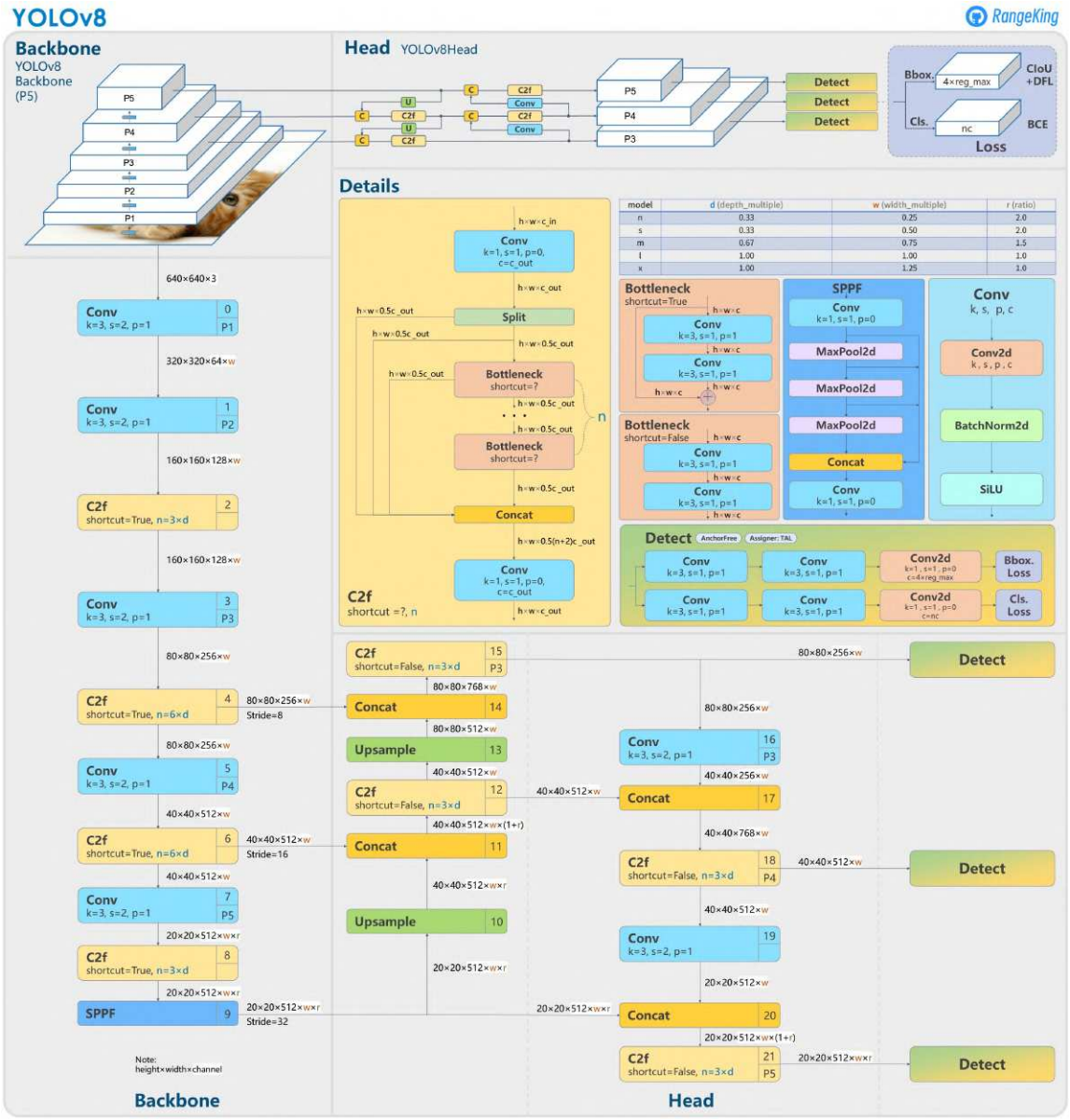

Figure 2.10: Model structure of YOLOv8 detection model [7].

2.5 Proportional Integral (PI) controller

The PI controller is a widely used control algorithm in the control sector due to its relatively simple structure, ease of understanding, and implementation [28]. The transfer function for a PI controller (Figure 2.11) can be defined in terms of the error $E(s) = R(s) - Y(s)$ and the controller output $U(s)$. [8], as defined by Eq. 2.7:

$$C_{PI}(s) = K_p + \frac{K_I}{s} \tag{2.7}$$

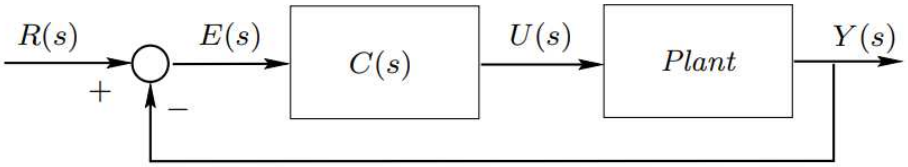

Figure 2.11: Basic feedback PI control loop [8].

where C_{PI} is the PI controller transfer function, K_P is the proportional gain and K_I is the integral gain. PI tuning is usually considered in terms of the P and I parameters, and although their effects in a closed loop plant are not independent of each other, they contribute as follows:

- Proportional component: its contributions depends on the instantaneous value of the control error. P controllers can control stable plants, but they provide limited performance and nonzero steady state error.
- Integral component: it provides a controller output proportional to the accumulated error, implying a slower reaction control mode. It helps achieve perfect plant inversion at $w = 0$, forcing the steady state error to zero with a step reference and disturbances.

Chapter 3

Methodology

In this chapter, the detailed approach to develop and implement a MAV autonomous landing system is presented. This landing system combines the use of YOLO v8 object detector as a landing marker detector, a MLP for drone-marker distance estimation, a PI control routine to control the drone position and ROS as the framework to control the drone and implement the algorithms previously mentioned. In Figure 3.1, the general road-map is presented.

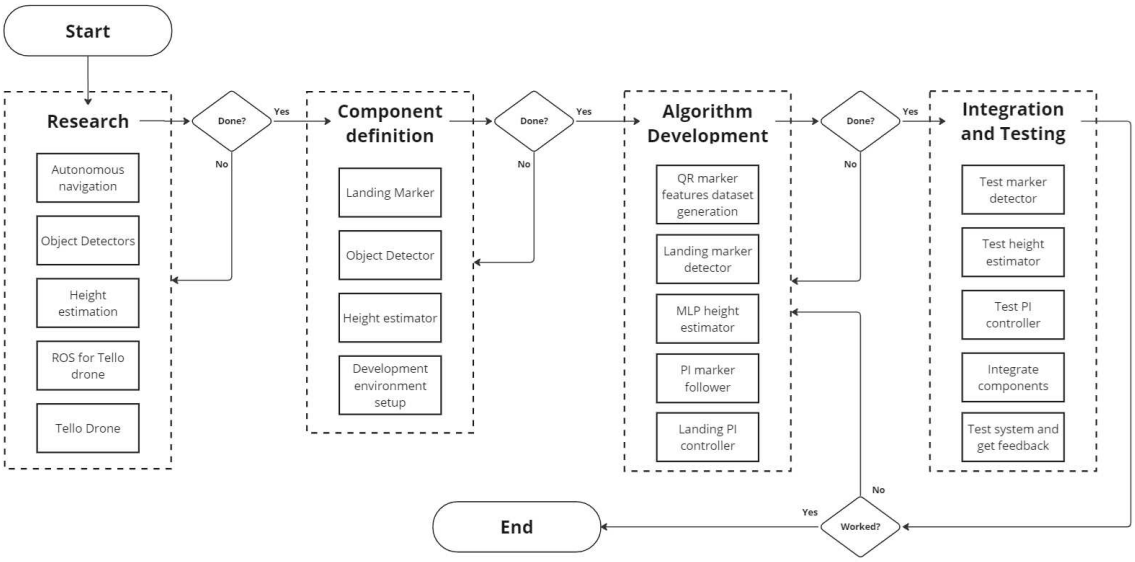

Figure 3.1: MAV autonomous landing general roadmap.

3.1 Component definition

After conducting research on autonomous navigation and autonomous landing methods of drones, the following equipment and approach were defined:

- DJI Tello drone for testing.
- Laptop with Rizen 7 CPU, Radeon Graphics integrated GPU, Ubuntu 20.04.5. This equipment was chosen due to a connection constraint with the Tello drone, discovered during the development environment setup stage. Initially, a laptop with a Ryzen 5 CPU and NVIDIA GeForce RTX 3060 GPU was intended to be used, however, the drone was not able to establish a connection with video feed to this equipment.
- 10x10 cm QR code as a landing marker, containing the string '10x10'.
- YOLOv8 as object detector (marker detector).
- MLP as height estimator.
- PI controller for landing routine.
- ROS Noetic as framework to implement the algorithms and control the drone.
- Tello driver as interface drone-ROS.

3.1.1 Tello drone

The Tello MAV is an entry-level quadrotor with dimensions of 9.8 cm x 9.25 cm x 4.1 cm. It includes a 5-megapixel camera with a field of view of 82.6° and 720p video resolution. This MAV can fly up to a height of 30 meters, with a maximum distance of 100 meters, a maximum velocity of 8 m/s, and a maximum flight time of 13 minutes. It supports a 2.4 GHz 802.11n Wi-Fi connection [29].

The Tello drone includes a front-facing camera; however, the camera needs to be downward-facing to detect the landing marker and perform the landing. To achieve this, a 2x2.5 cm mirror angled at 45° was attached to the front of the drone, as depicted in Figure 3.2. As a result of adding the mirror, the image is flipped vertically (inverted along the y-axis). This inversion will need to be accounted for in the development of the position control algorithm in the subsequent stages.

Figure 3.2: Tello drone equipped with a 2x2.5 cm mirror angled at 45° to direct the camera's view downward.

3.1.2 Development environment setup

With the research conducted previously and the software components defined, the next steps were followed to setup the development environment and to create the ROS workspace. A step to step tutorial can be consulted in Appendix 1.

- Install Ubuntu 20.04.5.
- Install python 3.8
- Install Visual Studio (VS) Code with python, C++ and CMake Tools extensions.
- Install ROS Noetic.

- Install dependencies for building packages.
- Create ROS workspace.
- Clone ROS-Tello driver [30] to workspace and install it.
- Compile workspace.

3.2 Algorithm Development

In this section, the development of the autonomous landing system is presented. In order to guarantee an optimal performance, a systematic approach was followed by developing and testing individually every component of the system before the final integration. It is important to mention that the Tello drone was used to test every script of the system.

3.2.1 Tello Driver

The first stage involved testing the Tello driver for ROS, identifying the primary constraint in the development of the detection and control algorithms. Initially, a laptop with an NVIDIA RTX 3060 GPU was planned for use; however, establishing a video feed connection with the Tello drone was unfeasible. Due to this limitation, the development environment was configured on a non-dedicated GPU laptop.

After establishing the connection Tello-ROS, the command rqt_image_view (from the library rqt, already integrated in ROS suite), was executed to verify that the video feed worked properly. In Figure 3.3, the connection diagram for Tello-ROS is presented. The driver (tello_driver_mod) publishes the video feed in /tello/image_raw ROS topic.

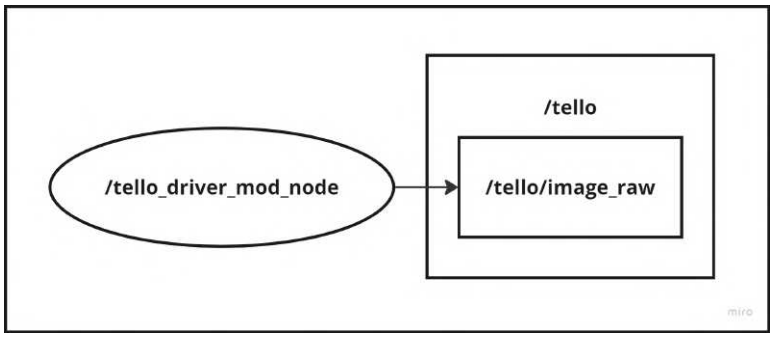

Figure 3.3: Tello-ROS connection diagram. The video feed is published on /tello/image_raw ROS topic.

3.2.2 Image interfacer module

In the previous step, the rqt module was used to test the video feed. However, to detect the QR landing marker, it is necessary to work directly with the frames received by ROS. ROS works with images in its own message format (sensor_msgs/Image), hence, to process the frames, it is necessary to interface ROS and OpenCV by converting ROS images to OpenCV images using cv_bridge, a library included in ROS framework. To implement this function, the script creates the node image_converter and subscribes to /tello/image_raw topic. Then, cv_bridge is used to convert the image, and finally, the video stream is shown in a window. This section of code will later be integrated into the marker detector and lander scripts. In Figure 3.4, a snippet of the code is presented, whereas in Figure 3.5, the Tello-ROS connection diagram is shown.

```
def callback(self,data):
    # Convert ROS Image message to cv2 format
    try:
        cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
    except CvBridgeError as e:
        print(e)

    # Convert image to gray scale
    gray_image = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

    # Show image in window
    cv2.imshow("Image window", gray_image)
    cv2.waitKey(3)
```

Figure 3.4: Snippet of the ROS-cv2 image interfacer module.

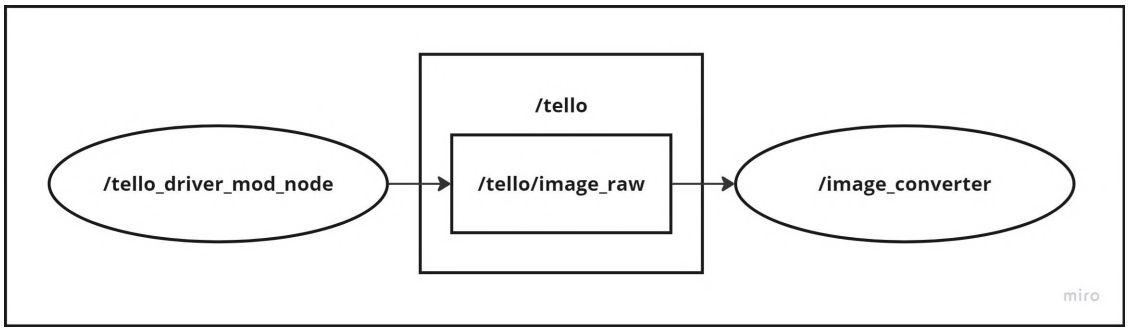

Figure 3.5: Tello-ROS connection diagram for image interfacer module.

3.2.3 Manual controller module

The next step consisted in the creation of a script to manually control the Tello drone, using the keyboard. It will serve to take-off the drone and move it to a position near the landing marker, where the autonomous mode will be activated to start the landing system. This script publishes the corresponding messages to different ROS topics to control the take-off, landing, movement, autonomous and manual mode of the drone, according to the Table 3.1.

After writing the keyboard control script, it was tested in conjunction with the image interfacer module, in order to verify the correct integrated movement control and image acquisition of the drone. In Figure 3.6, the Tello-ROS connection diagram is presented, along with the scripts running and the video feed obtained by the image interfacer.

3.2.4 Landing marker detector - height estimator module

This module has the function of detecting the selected QR code (Figure 3.7) using YOLOv8 object detector and estimating the height between the drone and the landing marker using a MLP. Then it publishes the center coordinates of the bounding box, the bottom horizontal line coordinates of the quad segmentation (of the QR) and the estimated height of the drone. This data will be used later by the landing controller to follow the QR marker and land over it.

Action	Message Type	Msg Components	Topic	Key
Takeoff	Empty	N/A	/tello/takeoff	T
Landing	Empty	N/A	/tello/land	Space
Move Forward	Twist	linear.x = speed	/tello/cmd_vel	W
Move Backward	Twist	linear.x = -speed	/tello/cmd_vel	S
Move Left	Twist	linear.y = speed	/tello/cmd_vel	A
Move Right	Twist	linear.y = -speed	/tello/cmd_vel	D
Ascend	Twist	linear.z = speed	/tello/cmd_vel	Up Arrow
Descend	Twist	linear.z = -speed	/tello/cmd_vel	Down Arrow
Rotate Left	Twist	angular.z = speed	/tello/cmd_vel	Left Arrow
Rotate Right	Twist	angular.z = -speed	/tello/cmd_vel	Right Arrow
Autonomous Mode	Int8	data = 5	/keyboard/override	X
Manual Mode	Int8	data = 10	/keyboard/override	C

Table 3.1: Actions and their corresponding messages published by the keyboard control script.

QR landing marker detector

In order to detect the QR code used as landing marker, QRDet, a pre-trained QR detector model based on YOLOv8 is utilized [31]. This YOLO v8 QR code detector takes an input image and returns the detection confidence, the coordinates of the corners of the bounding box, the coordinates of the center of the bounding box and the four corners polygon coordinates that segments the QR. In Figure 3.8 the QR code segmented and surrounded by a bounding box can be observed. Also, the center of the QR is plotted.

Height estimator

To estimate the distance between the drone and the landing marker, a MLP was trained. To this end, the data returned by the YOLOv8 QR detector was used: QR size, QR code area

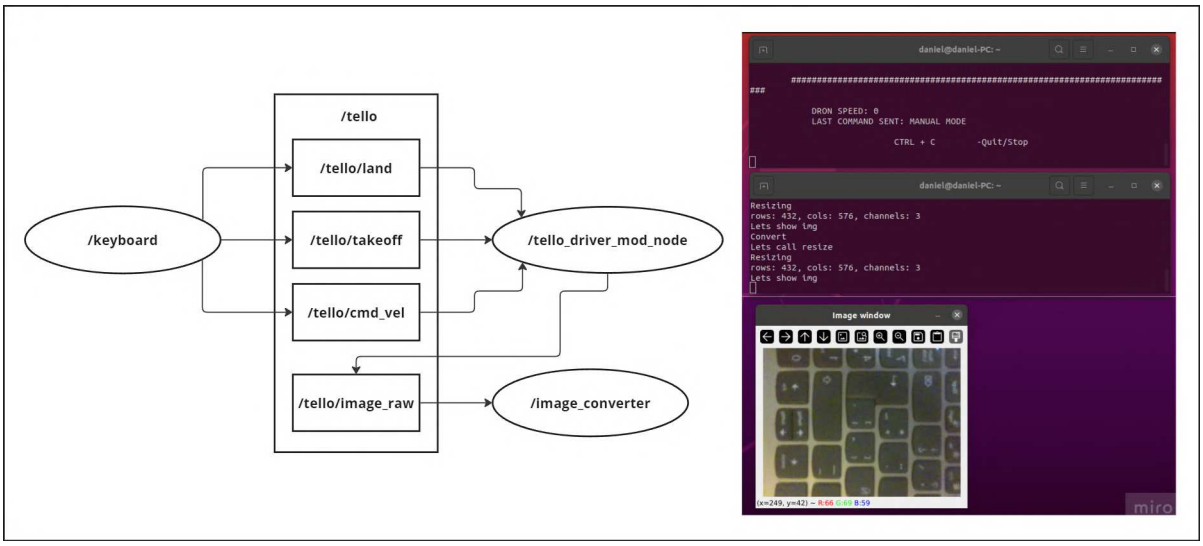

Figure 3.6: Tello-ROS connection diagram for keyboard and image interfacer modules.

Figure 3.7: 10x10 cm QR code selected as landing marker.

(computed with the bounding box corners coordinates) and x, y coordinates of the center of the bounding box. For the training dataset, the drone was flown to different heights (5, 10, 20, ... , 100, 110, 120 cm), and for every height a txt file was generated with the previously mentioned attributes.

Prior to the training, the data was analyzed and the outliers were deleted, then the 13 different heights were transformed to classes (with the labels 0 to 12) and one-hot encoding

Figure 3.8: QR marker detected with YOLOv8 QRDet. The QR is segmented and surrounded by the bounding box.

was performed. Also, the attributes were scaled. Using Keras, the next MLP architecture was generated:

- Input layer with 64 neurons and ReLU as activation function.
- Second layer with 32 neurons and ReLU as activation function.
- Output layer with 13 neurons, using Softmax activation function to generate the class probability.
- Adam optimizer.
- Categorical crossentropy as loss function.
- 1300 epochs.
- Batch size = 32.
- Training size = 0.8, test size = 0.2.

Both submodules were tested individually and then integrated in the script. The script subscribes to the topic /tello/image_raw to get the video frames and generates the node image_converter, publishing the following topics (Table 3.2), which will be used by the lander module:

Topic	**Message Type**	**Function**
/qr_detection/center	Int32MultiArray	Publishes the center coordinates (X, Y) of the bounding box of the detected QR marker.
/qr_detection/quad_points	Int32MultiArray	Publishes the coordinates of the points forming the bottom edge of the quadrilateral segmented from the QR marker.
/qr_detection/predicted_height	Int32	Publishes the estimated height of the drone relative to the QR marker.

Table 3.2: Topics published by the QR detector - height estimator script.

The Tello-ROS connection diagram for the QR detector - height estimator module is depicted in Figure 3.9, alongside a screenshot of the camera image and the console log displaying the predicted height, center coordinates, bounding box (depicted as a black box), bounding box center (represented by a black point at the center), and the segmentation (depicted as a white quadrilateral). It's important to note that the diagram does not include the topics published by /image_converter, as no node is currently subscribed to them.

3.2.5 Lander module

Finally, the lander module script was programmed. This module receives data published by the QR detector-height estimator module and uses it to calculate position errors in the x, y, and z axes. The PI controller then computes the output signals for the drone and publishes them to the corresponding ROS topics.

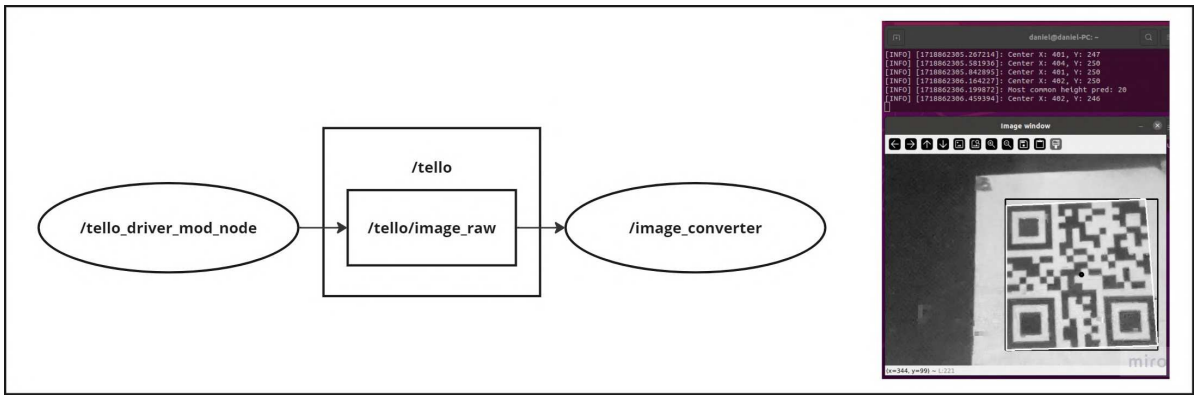

Figure 3.9: Tello-ROS connection diagram and screenshot of the script detecting and segmenting the QR marker.

PI controller

In order to obtain the control signal for x, y z axis and yaw, the following equations are followed:

$$U_i = k_{pi} * e_i + k_{ii} * \int e_i * dt \tag{3.1}$$

And the discrete calculation is:

$$U_i = k_{pi} * e_i + k_{ii} * \sum_{i=1}^{n_i} e_i \tag{3.2}$$

Where U_i is the output signal control for x, y, z and yaw, k_{pi} are the proportional gains, k_{ii} the integral gains, $\int e_i * dt$ is the accumulated (integral) error for axis i. The integral error is approximated by summing the error at each discrete time step: $\sum_{i=1}^{n_i} e_i$.

To calculate the position errors for x, y and z axis, the lander module subscribes to the topics /qr_detection/center (which contains the coordinates of the center of the bounding box of the detected QR) and /qr_detection/predicted_height (the height of the drone with respect to the landing marker, predicted by the MLP). Then the errors are calculated as follows:

$$e_x = x_q - x_c, \qquad e_y = y_q - y_c, \qquad e_z = 0 - height \tag{3.3}$$

Where x_q and y_q are the coordinates of the center of the QR landing marker, x_c and y_c are the coordinates of the desired position (the center of the coordinate plane), 0 is the desired position in z axis (landing platform) and height is the height predicted by the MLP. In Figure 3.10, the coordinate plane of the video feed is represented. It is important to note that the image is flipped along the y-axis.

On the other hand, to calculate the yaw error, the lander module subscribes to the topic /qr_detection/quad_points, which contains the coordinates of the points forming the bottom edge of the quadrilateral segmented from the QR marker. These coordinates are converted into a vector and normalized. Additionally, a horizontal vector is created and normalized. The dot product between these vectors is then calculated, and the angle is determined by taking the arccosine of the resulting dot product. Finally, the angle obtained is converted from radians to degrees. Figure 3.11 presents a representation of this operation.

Finally, the gains are set heuristically and the output control signals are calculated, then they are published to the corresponding topics. It is important to note that the drone's x and y axes do not correspond to the camera coordinate plane defined in the previous subsections, as they are inverted. Therefore, the control signal U_x (right-left movement) is sent in the $linear.y$ component of the Twist message published to the $/tello/cmd\_vel$ topic, and the control signal U_y (forward-backward movement) is sent in the $linear.x$ component. The output control signals for the z axis and yaw are sent in $linear.z$ and $angular.z$, respectively. On the left side of Figure 3.12, the Tello-ROS connection diagram is depicted, showing the outputs of the QR follower published to $/tello/cmd\_vel$ and $/tello/land$. On the right side of the same Figure, the lander script is shown calculating the position errors, along with the drone's video feed displaying the segmented QR and its bounding box.

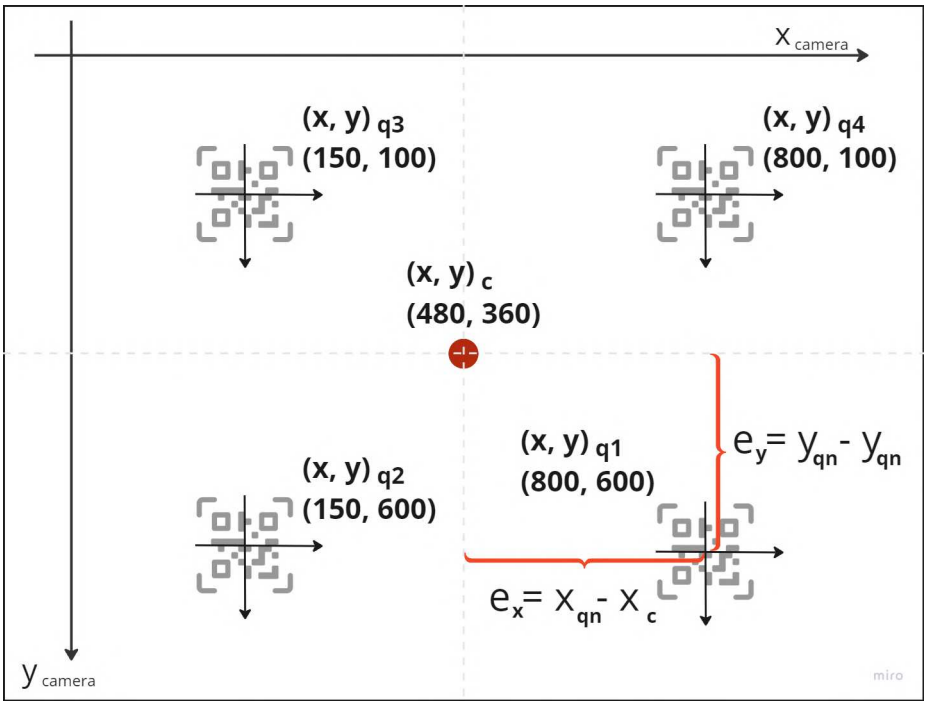

Figure 3.10: Representation of the coordinate plane for the video feed, illustrating the flipped y-axis. This setup is used by the lander module script to calculate position errors for precise drone landing.

3.2.6 Integration and Testing

Once all of the scripts of the system were tested individually, the integrated system was tested as follows:

- Landing marker detector and manual controller modules: Configuration tested to verify the correct on-flight landing marker detection.
- Landing marker detector - height estimator and manual controller modules: Configuration tested to verify on-flight landing marker detection and height estimation.
- Landing marker detector and lander modules: Configuration tested to verify on-flight landing marker detection and the PI position controller over the x and y axes. This test was divided into two parts: positioning of the drone over the center of the fixed landing marker and positioning of the drone over the center of the landing marker in motion.

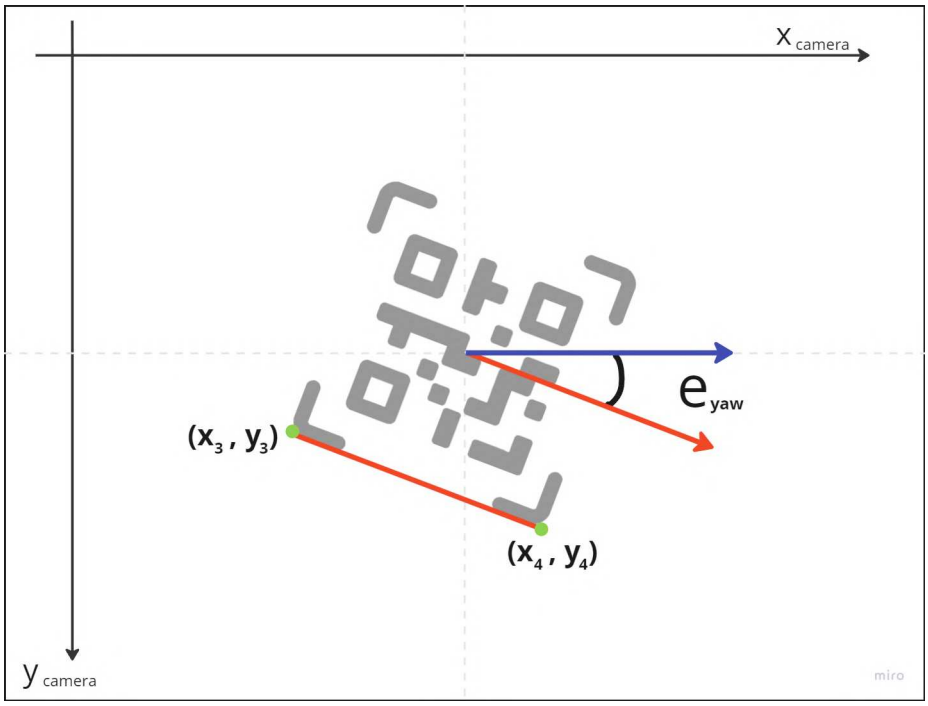

Figure 3.11: Representation of the yaw error calculation process. The vectors formed by the bottom edge of the quadrilateral and a horizontal reference are normalized, their dot product is computed, and the angle between them is obtained by taking the arccosine of the dot product result, finally converting it from radians to degrees.

- Landing marker detector - height estimator and lander module: Configuration tested to verify the correct autonomous landing of the drone over the QR marker. This test was divided into two parts: landing over a fixed marker and landing over a marker on a moving platform.

In order to run the integrated system with the Tello drone, the following commands and scripts are executed on individual linux terminals:

- roscore ## Execute master node
- rosrun driver_tello_mod tello_driver_mod.py ## Execute tello driver node
- rosrun keyboard tello_keyboard.py ## Start keyboard control node

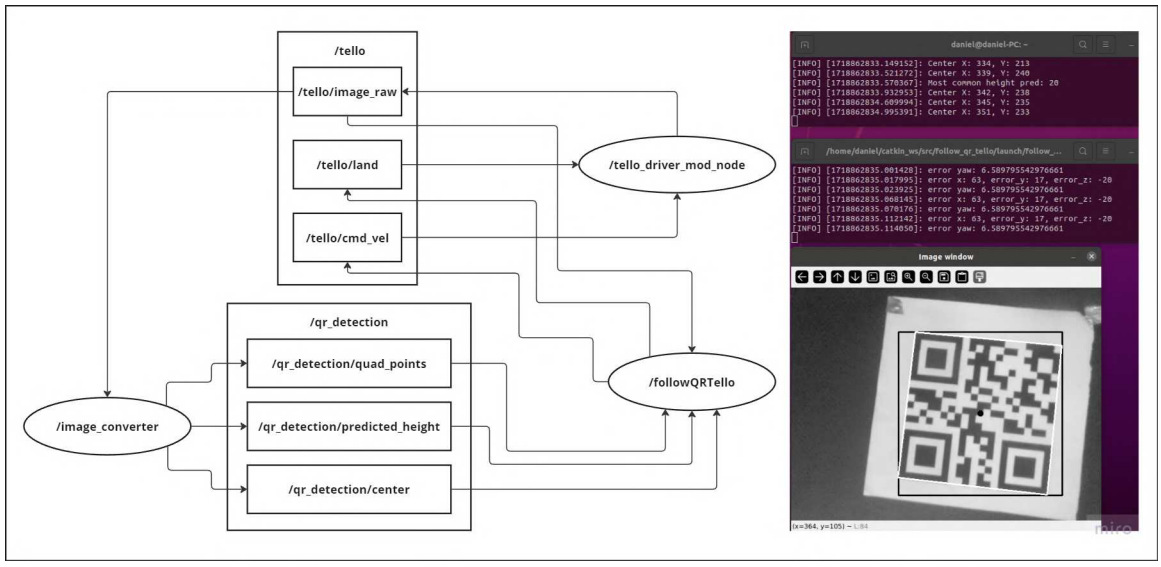

Figure 3.12: Tello-ROS connection diagram and lander script with video feed, showing position error calculation, QR segmentation and bounding box.

- rosrun qr_detector_tello qr_detector.py ## Start qr_detector node
- roslaunch follow_qr_tello follow_qr_tello.launch image_reduction:=60 ## Start qr_follower node

Using the manual mode of the manual controller module, the drone is taken-off and positioned over the QR landing marker. When a portion of the QR is visible in the camera, the autonomous mode is activated and the autonomous lander starts working. In Figure 3.13, the Tello-ROS connection diagram is depicted, along with the QR detector - height estimator and lander modules, showing the marker detection, the position error calculation, the control signals sent to the drone, and the video feed.

During the testing stage, minor adjustments to the gain values of the PI controller were done. The results of the integrated system test are presented in next chapter.

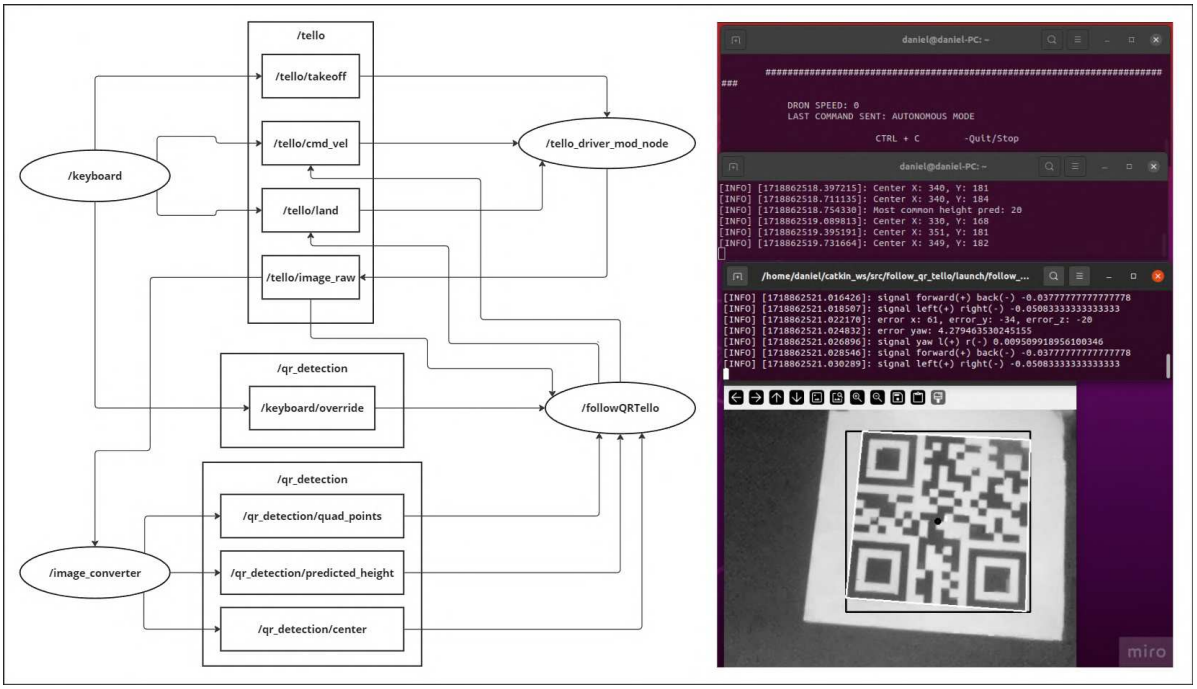

Figure 3.13: Tello-ROS connection diagram with QR detector - height estimator and lander modules running.

Chapter 4

Results

The autonomous landing system was evaluated in different configurations of the modules to verify the correct detection of the landing marker, the precise height estimation and the effective working of the PI position controller to achieve a safe landing on a moving platform.

4.1 Landing marker detection

YOLOv8 QRDet model, adapted for on-flight landing marker detection, showed a high performance on the identification of the QR marker under different light conditions and camera angles. It's important to remark that it was able to detect incomplete QR codes, which is an important feature since the lander module can start the position controller before the marker is complete on the field vision of the camera. Prior to test the detector with the drone on flight, it was benchmarked with a dataset of images containing QR codes and random surface textures. After that, the next metrics and plots were obtained: confusion matrix, precision, recall, f1-score, accuracy and precision vs recall plot.

A confusion matrix is a tabular way to visualize the performance of the neural network. Each cell in a confusion matrix indicates quantity of predictions generated by the model and shows whether it classified the classes accurately or inaccurately. The generated matrix is

shown in Figure 4.1.

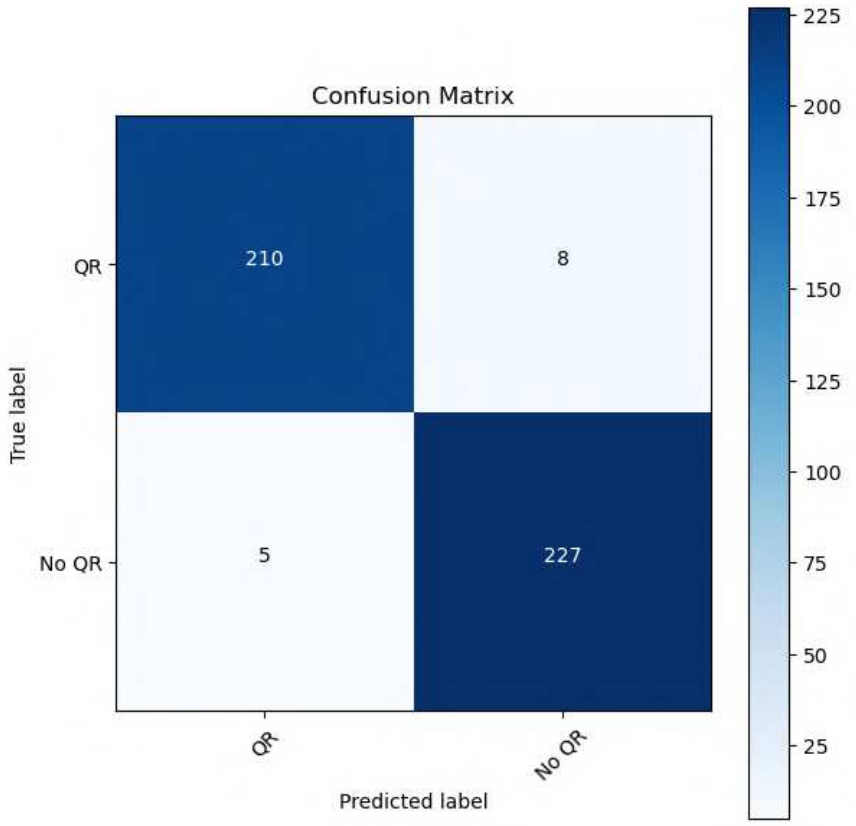

Figure 4.1: Confusion matrix of YOLO QR detector evaluated with the test dataset.

Similarly, the metrics for the QR detections are calculated from the confusion matrix, and each one answers the following questions, supporting the evaluation of the QR detector model:

- Precision: Of all predicted positives, what percentage is truly positive? This value is obtained using the following equation:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{4.1}$$

- Recall: Of all true positives, what percentage was predicted as positive? This metric is obtained using the following equation:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{4.2}$$

- F1-score: It is the harmonic mean of precision and recall, considering both false positives and false negatives. Therefore, it performs well on imbalanced datasets [32], meaning when there is an unequal distribution of classes in the training dataset. This value is obtained as follows:

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.3}$$

- Accuracy: Provides the overall accuracy of the model, i.e., the percentage of total samples that YOLO detected correctly. The equation is:

$$Accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \times \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i) \tag{4.4}$$

where $\hat{y}_i$ is the predicted value of the i-th sample (from the validation set of size $n_{samples}$), y_i is the corresponding true value, and 1(x) is the indicator function.

In Figure 4.2, the metrics obtained from the benchmarking of YOLO QR detector are presented, where the class '0' is 'No QR' and '1' is 'QR' 4.2.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.96      0.97       218
           1       0.97      0.98      0.97       232

    accuracy                           0.97       450
   macro avg       0.97      0.97      0.97       450
weighted avg       0.97      0.97      0.97       450

Precision: 0.97
Recall: 0.98
F1 Score: 0.97
Accuracy: 0.97
```

Figure 4.2: Performance metrics for YOLO QR detector evaluated with the test dataset.

The performance of the QR detector was evaluated in terms of the precision and recall, considering the previous definitions and the following requirements [33]:

- Precision: It is important that during the landing routine all of the detections be true positives, since a false positive detection (identify a non-QR as QR marker) could lead to the PI controller to move the drone to a non-desired position, trying to center and land the drone over a non-QR marker, or slowering the landing process. Therefore, a 0.97 precision value indicates that the detector is highly reliable and it's highly probable that no false positives will be detected, achieving a fast and safe landing over the landing marker.

- Recall: It is required that the QR marker be detected during the whole landing process, since a false negative detection (identify a QR marker as non-QR) could led the PI controller to stop the landing routine, making it slower. Also, since the landing is on a moving platform, a series of false negative detections could cause that the PI controller stops chasing the marker and it be out of the camera's field of view, aborting the landing. Hence, a recall of 0.98 indicates that it's highly improbable that a false negative detection occurs, achieving a successful chasing and landing over the moving platform.

Finally, a Precision vs Recall curve was plotted (Figure 4.3), showing that the QR detector minimizes both, the false positives and false negatives, and hence it's reliable for the QR marker detection during the landing task.

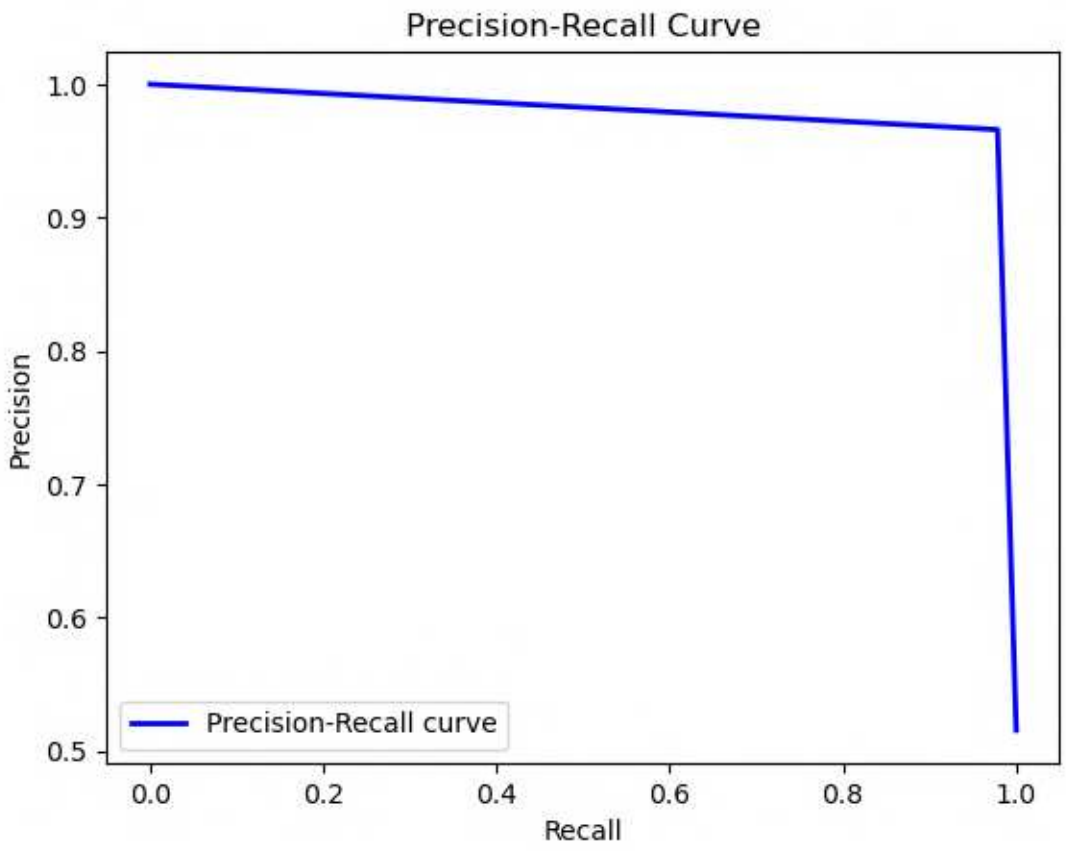

Figure 4.3: Precision vs Recall curve, YOLOv8 QR detector minimizes false positives and false negatives.

4.2 Height estimator

As mentioned in Methodology chapter, the height estimator MLP was trained with data obtained from the YOLO QR detector, associated to the height where the data points were captured. The MLP estimator was tested with a part of the dataset generated, obtaining a good to high precision and recall in the estimation (Figure 4.4), depending on the height of the drone. According to the metrics, the model tends to estimate the height correctly when the drone is nearer to the QR marker, and it fails more when it is flying higher. This is caused by the size difference if the bounding box (used by the MLP), the higher the drone flies, the smaller the difference in the bounding box area for every height prediction.

	precision	recall	f1-score
5	1.00	1.00	1.00
10	1.00	1.00	1.00
20	1.00	1.00	1.00
30	1.00	1.00	1.00
40	1.00	1.00	1.00
50	0.94	0.94	0.94
60	0.89	0.73	0.80
70	0.90	0.97	0.93
80	0.88	0.88	0.88
90	0.89	0.80	0.84
100	0.69	0.82	0.75
110	1.00	0.67	0.80
120	0.80	0.80	0.80
accuracy			0.93
macro avg	0.92	0.89	0.90
weighted avg	0.93	0.93	0.93

Figure 4.4: Metrics obtained in the evaluation of the MLP.

Also, he Mean Squared Error (MSE) was used to evaluate the performance of the MLP estimator. The MSE is a metric used to measure the average of the squares of the errors between the actual values and the predicted values by a model. It quantifies the difference between the values predicted by a model and the actual values, providing an indication of the model's accuracy. The formula for MSE is given by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.5}$$

where:

- n is the number of observations.
- y_i represents the actual value.
- $\hat{y}_i$ represents the predicted value.

A lower MSE indicates that the predicted values are closer to the actual values, suggesting a more accurate model. The MSE was calculated with the MLP test set, obtaining a value of 13.13, which is good enough considering that the range of estimation id from 5 cm to 120 cm. A comparison between the real height and the estimated height is presented in Figure 4.5, showing also the calculated MSE.

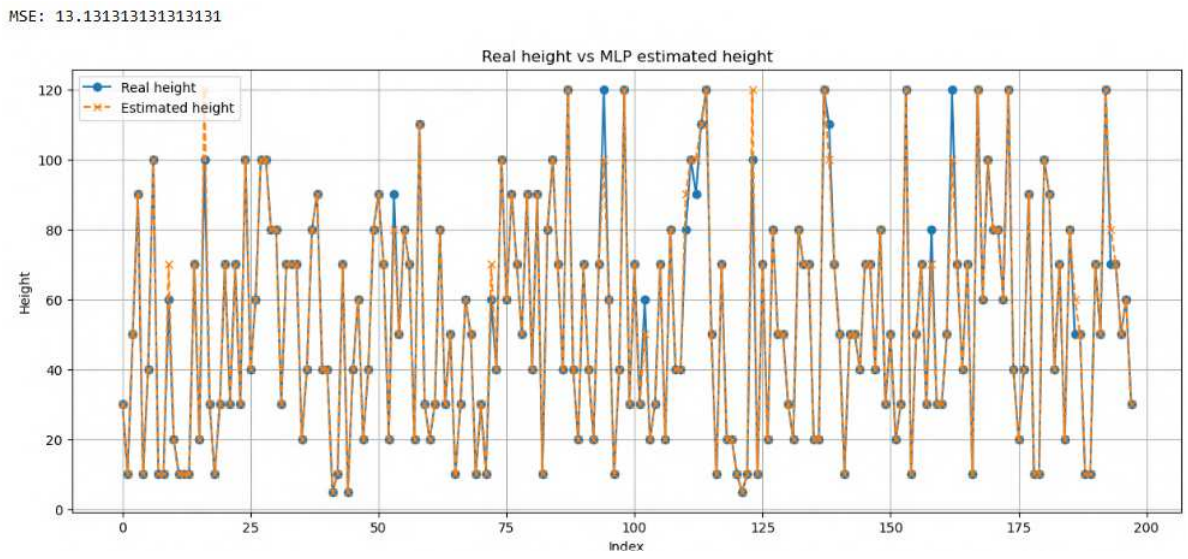

Figure 4.5: Real height vs the MLP estimated height.

4.3 Integrated system

The test of the integrated system on the Tello drone was performed in a controlled environment. The scripts and process was executed as described in Methodology chapter. 20 flights were performed, obtaining the results presented in Table 4.1. Distance indicates the distance the drone traveled (starting when it was positioned near the QR marker until it landed), landing error (e_x, e_y) indicates the position error in landing in centimeters (distance between the center of the drone to the center of the QR marker for x and y axis), time to land is the time taken to land in seconds (measured since the positioning of the drone near the QR marker), FPS is the number of evaluations per second performed by the system (detect QR and its position, estimate height and use PI controller to send control signals to Tello drone) and Status indicates if the drone achieved a successful landing.

Test Number	**Distance (m)**	**Landing Error (e_x, e_y) (cm)**	**Time to Land (s)**	**FPS**	**Landing Status**
1	1.2	(1, 1)	9	4	Success
2	1.5	(2, 0)	10	4	Success
3	2.0	(3, 2)	12	4	Success
4	2.2	(1, 1)	13	4	Success
5	1.8	(4, 3)	11	4	Success
6	-	(-, -)	-	4	Fail
7	2.1	(1, 1)	13	4	Success
8	1.3	(0, 2)	9	4	Success
9	1.7	(1, 1)	11	4	Success
10	1.9	(5, 4)	12	4	Success
11	2.4	(1, 0)	15	4	Success
12	1.4	(2, 1)	9	4	Success
13	2.3	(1, 5)	14	4	Success
14	1.6	(0, 0)	10	4	Success
15	2.0	(1, 1)	12	4	Success
16	2.5	(3, 2)	16	4	Success
17	1.1	(1, 1)	8	4	Success
18	-	(-, -)	-	4	Fail
19	2.3	(1, 1)	15	4	Success
20	2.5	(0, 0)	16	4	Success
Average	1.88	(1.56, 1.44)	11.94	4	-

Table 4.1: Test results for autonomous landing system.

In Figures 4.6 and 4.7, two successful autonomous landing test are depicted, for easy identification of the position of the drone and moving platform, they are surrounded with a red and green boxes, respectively. In Figure 4.6, the test was performed with the platform moving in a flat-linear path, while the Figure 4.7 test was performed on a path with a slope in

the initial section, demonstrating the capability of the QR detector to detect a QR in different angles.

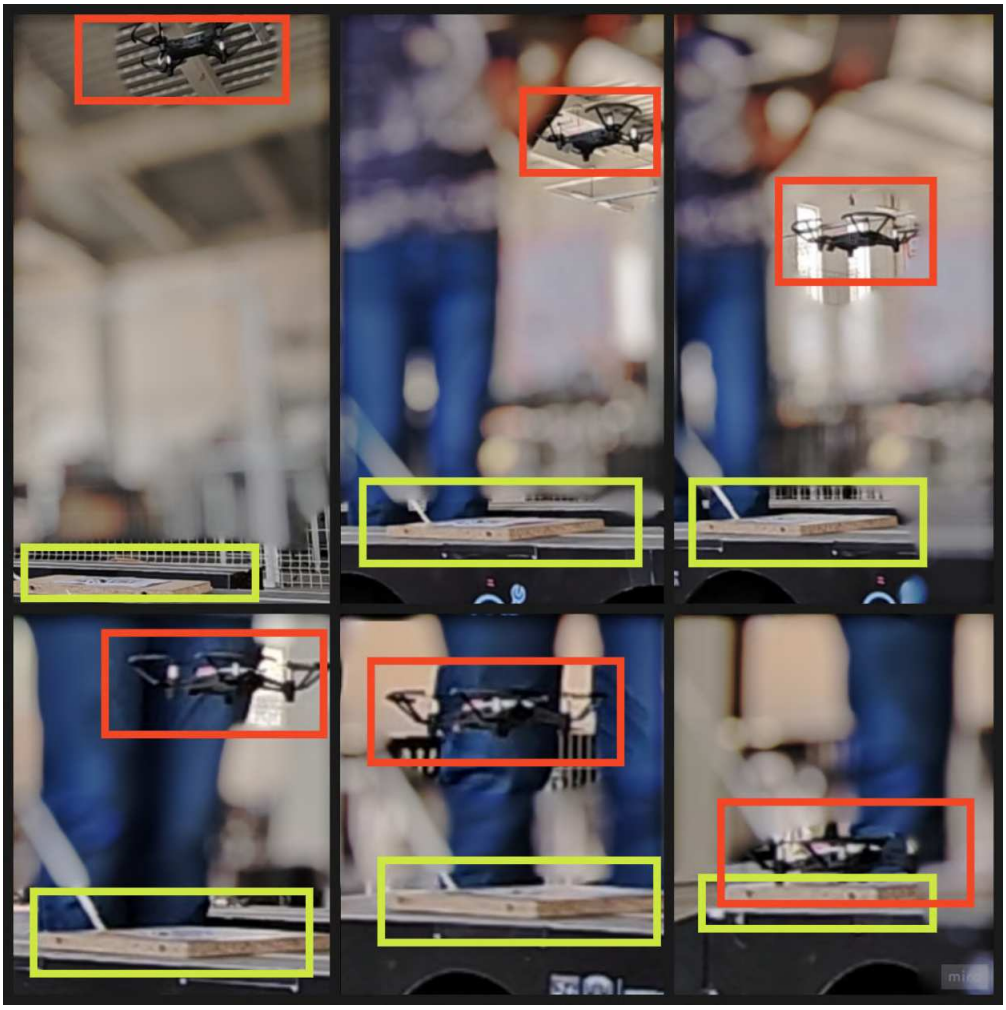

Figure 4.6: Test of the autonomous landing system on a flat path.

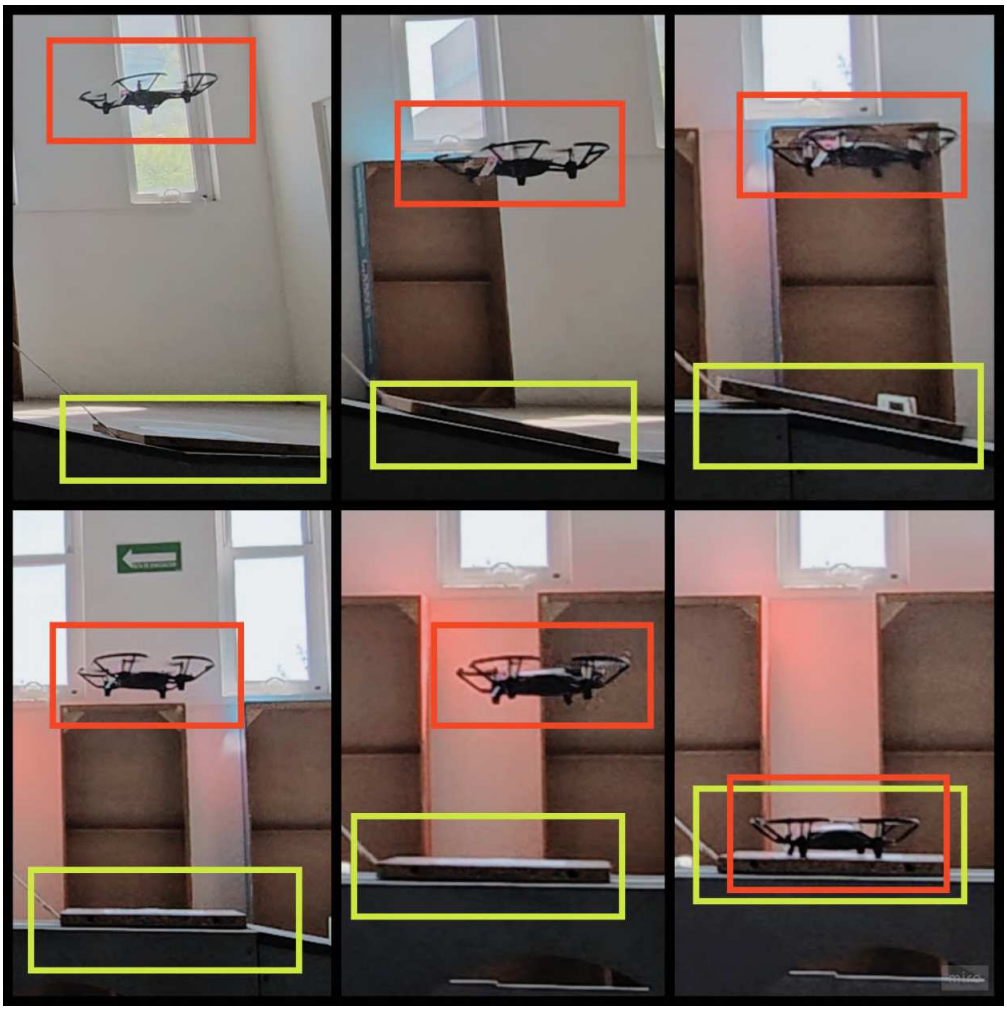

Figure 4.7: Test of the autonomous landing system on a irregular path.

As shown in Table 4.1, the autonomous landing system performs well, having a landing success rate of 90%, with an average landing error of 1.5 cm and 1,4 cm for x and y axis, with an average time to land of 11.94 s and an average landing distance of 1.88 m. Hence, the average velocity of the drone during the landing is $0.15\frac{m}{s}$. The system performs 4 evaluations per second (4 FPS) when running all the modules, but reaches up to 21 FPS when only landing marker detector - height estimator module is executed. The time to land is highly affected by the technical limitations of the laptop used for processing, since it takes more time to process the video feed frames, and the YOLO and MLP models due to the lack of a dedicated GPU.

In Tables 4.2 and 4.3, a summary of the autonomous landing systems is presented. The proposed method demonstrates competitive accuracy compared to other algorithms, noting that it is surpassed or matched by approaches based on two or more combined algorithms, with the exception of SSD. This is possibly due to the fact that YOLOv8 QRDet was evaluated on a dataset that contained, a priori, images similar to QRs, such as ARUCO markers. While its efficiency is lower in terms of FPS, this is due to hardware limitations. Despite this drawback, the system has been able to perform precise landings on a moving platform, with a landing position error of less than 2 cm, demonstrating its reliability in dynamic environments. Despite these limitations, the system shows solid performance and has the potential to improve its efficiency with more advanced hardware.

Author(s)	**Contribution**	**AI Technique**	**Dataset**
Yu et al.	Robustness under different conditions	CNN + SqueezeNet (own architecture)	Synthetic simulated scenarios and real-world videos
Nguyen et al.	Remote-marker-based tracking algorithm using visible-light-camera under different environment conditions	lightDenseNet + YOLOv2 + Profile Checker v2	Markers taken from long and close distances
Lin, Jin, and Chen	Monocular vision system for landing marker localization at night	CNN + Decision Tree	Own-generated dataset of landing markers and segmented natural scenes

Author(s)	**Contribution**	**AI Technique**	**Dataset**
Cabrera et al.	Detection system of landing mark for autonomous landing using a portable computer with NVIDIA GPU (onboard)	Single Shot Detector (SSD)	5000 320x240 pixels RGB images
Wu et al.	Autonomous landing on a mobile platform using PID and DDPG	Deep Reinforcement Learning (DDPG)	Gazebo virtual environment
Piponides et al.	Camera sensor-based landing system with RL controller	DroNet detector and PPO algorithm	3419 real world images
This research	Monocular camera based, YOLOv8 features (detection, segmentation) as core of the system, robust marker detection, PI controller, landing on moving platform	YOLOv8 QRDet, MLP height estimator	Marker data obtained from YOLO detections and segmentations

Table 4.2: Summary of methodologies and and their contributions.

Method	**Accuracy (%)**	**Success Rate (%)**	**Landing Error (cm)**	**Detector FPS (PC - Onboard)**
YOLO + SqueezeNet	83.7	N/A	(8.2, 9.11)	(N/A - 21)
lightDenseNet + YOLOv2 + Profile Checker v2	99.0	N/A	N/A	(40 – 20)
CNN + Decision Tree	97.0	N/A	N/A	(40 - 12)
SSD	98.0	N/A	N/A	(90 – 13)
PID + DDPG	N/A	97.0	N/A	N/A
DroNet + PPO	95.0	75.0	N/A	N/A
YOLOv8 + MLP	97.0	90.0	(1.5, 1.4)	(21, N/A)

Table 4.3: Comparison of methodologies and the metrics used for evaluation.

The main contribution of this research is the use of YOLOv8 as core of the systems, since the detection and segmentation features are used to feed the MLP height estimator and the PI position controller, helping also to achieve robustness during the detection, hence, in all the depending modules. In future work, landing velocity has a wide range of improvement.

Chapter 5

Conclusions

This research presented the development of an autonomous landing system for Micro Aerial Vehicles (MAVs), commonly known as micro drones, to land safely on a moving platform, successfully achieving all proposed objectives and goals. An extensive review of the state of the art was conducted as the first step to familiarize with the approaches, algorithms, and technologies commonly used.

Taking this research as starting point, a landing system based on YOLOv8 was proposed, utilizing the Tello drone's camera as unique sensor, thus avoiding additional payload. The use of a pre-trained YOLOv8 QR detector resulted in a significant improvement in the precision of landing marker detection, proving to be equal or more robust than other state-of-the-art systems. This enhancement played a crucial role in the autonomous landing routine, ensuring that as long as the QR marker is within the drone's field of vision, the marker's position would not be lost, allowing the drone to accurately track it.

The native features of YOLOv8 were leveraged to train and feed an MLP height estimator. The output from this estimator, combined with the native features, served as inputs for the PI position controller, enabling a successful landing on the moving platform.

The ROS framework facilitated the design and implementation of the system's modules, allowing for individual development and rapid integration. Tests conducted with the Tello drone and the integrated modules demonstrated that the landing system could continuously detect the QR landing marker, estimate the height, follow a moving platform, and land on it with high precision and within an acceptable timeframe.

With all objectives achieved, this system presents a viable alternative for autonomous landing in GPS-denied environments, proving to be valuable in scenarios where a pilot cannot access, such as delivering first aid equipment during natural disasters like floods, requiring only a QR marker for landing.

Finally, further research may be needed to enhance height estimation and landing velocity by exploring different algorithms and utilizing a computer with a dedicated GPU or an onboard computer.

Chapter 6

Appendix 1: ROS and Tello Setup

6.1 ROS Installation

http://wiki.ros.org/noetic/Installation/Ubuntu

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >

sudo apt install curl # if you haven't already installed curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-ke

sudo apt update
sudo apt install ros-noetic-desktop-full

source /opt/ros/noetic/setup.bash
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc

sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator pyth
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

6.2 Create Workspace

http://wiki.ros.org/catkin/Tutorials/create_a_workspace

```
In terminal:
mkdir -p ~/tello_ws/src
cd ~/tello_ws
catkin_make

In home directory:
gedit .bashrc
Add to bashrc:
source /home/daniel/tello_ws/devel/setup.bash
```

6.3 ROS-Tello Installation

https://github.com/anqixu/tello_driver

```
# Install pip if not installed
sudo apt install python3-pip

## Create workspace if not created yet
mkdir -p ~/tello_ws/src
cd ~/tello_ws
catkin_make

# Clone repo to workspace
cd tello_ws/src
git clone https://github.com/anqixu/TelloPy.git

cd TelloPy
sudo -H pip3 install -e .

cd ..
git clone https://github.com/anqixu/h264_image_transport.git
git clone https://github.com/anqixu/tello_driver.git
```

```
cd ..
rosdep install h264_image_transport

## If previous command fails, run:
rosdep update
rosdep install --from-paths src -i
## If this also fails, do:
sudo cp -r ~/tello_ws/src/h264_image_transport /opt/ros/noetic/share/

catkin_make
source devel/setup.bash

pip3 install av

Go to:
catkin_ws/src/tello_driver/src
Open:
tello_driver_node.py

Modify the first line from:
#!/usr/bin/env python2
To:
#!/usr/bin/env python3

Turn on drone and connect to it

Open new terminal and run:
roscore
Open new terminal and run:
roslaunch tello_driver tello_node.launch
```

6.4 Create ROS Package

http://wiki.ros.org/ROS/Tutorials/CreatingPackage

In terminal (home):
cd tello_ws/src
catkin_create_pkg first_pkg std_msgs rospy roscpp

6.5 Install VS Code

https://code.visualstudio.com/docs/?dv=linux64_deb

Download VS Code

In download directory:
sudo dpkg -i code_1.86.0-1706698139_amd64.deb

Install extensions:
Python
C++/C
CMake Tools

6.6 Simple Publisher Node ROS

http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

In VS Code:
Open working folder (use File -> Open Folder option) "tello_ws":
In tello_ws/src/first_pkg/src:
New File (talker.py)
Write or copy publisher node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/first_pkg/src ## Go to package src folder (or publisher node folder)
chmod +x talker.py ## Give execution privileges to publisher node (talker.py)
cd ## Go to home

```
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project
## If error in compilation (related to std_msgs.msg package):
1 - Go to /home/daniel/tello_ws/src/first_pkg
2 - Open CMakeLists.txt and package.xml, then change all "std_msg" string to "std_ms
3 - Save changes in files and execute "catkin_make" in VS Terminal again, it should

Open new linux terminal and run:
roscore  ## Execute master

Open new linux terminal and run:
rosrun first_pkg talker.py   ## Execute publisher node

Open new linux terminal and run:
rostopic list   ## Shows all active topics, here must appear "/chatter" topic,
## as defined in rospy.Publisher('chatter', String, queue_size=10), in publisher scr

Open new linux terminal and run:
rostopic echo /chatter   ## Shows the data being published by publisher (talker.py)

Open new linux terminal and run:
rosnode list  ## Shows all active nodes, here must appear master node and publisher
```

6.7 Simple Subscriber Node ROS

```
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

In VS Code:
Open working folder (use File -> Open Folder option) "tello_ws":
In tello_ws/src/first_pkg/src:
New File (listener.py)
Write or copy subscriber node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/first_pkg/src ## Go to package src folder (or subscriber node folder
```

```
chmod +x listener.py ## Give execution privileges to subscriber node (listener.py)
cd  ## Go to home
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project
## If error in compilation (related to std_msgs.msg package):
1 - Go to /home/daniel/tello_ws/src/first_pkg
2 - Open CMakeLists.txt and package.xml, then change all "std_msg" string to "std_msgs"
3 - Save changes in files and execute "catkin_make" in VS Terminal again, it should compile n

Open new linux terminal and run:
roscore  ## Execute master

Open new linux terminal and run:
rosrun first_pkg talker.py   ## Execute publisher node

Open new linux terminal and run:
rosrun first_pkg listener.py   ## Execute subscriber node

Open new linux terminal and run:
rosnode list  ## Shows all active nodes, here must appear master node, publisher node ("/talk

Open new linux terminal and run:
rostopic list   ## Shows all active topics, here must appear "/chatter" topic,
## as defined in rospy.Publisher('chatter', String, queue_size=10), in publisher script (talk
## and in rospy.Subscriber("chatter", String, callback), in subscriber script (listener.py)

Open new linux terminal and run:
rostopic info /chatter ## Shows the type of message that is being sent (std_msgs/String),
## the publisher ( * /talker_10521_1706996427712 (http://daniel:43941/) )
## and the subscriber ( * /listener_10590_1706996441881 (http://daniel:43317/) )

Open new linux terminal and run:
rqt_graph  ## It will show the connection of the nodes
## IN some versions of ROS it runs with: rosrun rqt_graph
```

6.8 Keyboard Control for Tello

https://alfredo-reyes-montero.gitbook.io/tello-dji/frameworks/ros

```
Open new linux terminal and run:
cd tello_ws/src
catkin_create_pkg keyboard geometry_msgs roscpp rospy std_msgs   ## Create package w

Open new linux terminal and run:
pip3 install getkey   ## Package required to use keyboard to control the drone

In VS Code:
Open working folder (use File -> Open Folder option)  "tello_ws":
In tello_ws/src/keyboard/src:
New File (tello_keyboard.py)
Write or copy node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/keyboard/src ## Go to package src folder
chmod +x tello_keyboard.py ## Give execution privileges to node (tello_keyboard.py)
cd  ## Go to home
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project
## If error in compilation (related to std_msgs.msg package):
1 - Go to /home/daniel/tello_ws/src/name_pkg
2 - Open CMakeLists.txt and package.xml, then change all "std_msg" string to "std_ms
3 - Save changes in files and execute "catkin_make" in VS Terminal again, it should

Open new linux terminal and run:
roscore   ## Execute master node

Open new linux terminal and run:
roslaunch tello_driver tello_node.launch    ## Execute tello driver node

Open new linux terminal and run:
rosrun keyboard tello_keyboard.py  ## Start keyboard control node
```

6.9 Acquire Image from Tello Camera

https://alfredo-reyes-montero.gitbook.io/tello-dji/frameworks/ros http://wiki.ros.org/cv_bridge https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCV

```
Open new linux terminal and run:
cd tello_ws/src
catkin_create_pkg image_viewer_tello cv_bridge sensor_msgs image_transport roscpp rospy std_m
## Create package with dependencies to use

In VS Code:
Open working folder (use File -> Open Folder option) "tello_ws":
In tello_ws/src/image_viewer_tello/src:
New File (image_viewer.py)
Write or copy node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/image_viewer_tello/src ## Go to package src folder
chmod +x image_viewer.py ## Give execution privileges to node (image_viewer.py)
cd  ## Go to home
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project
## If error in compilation (related to std_msgs.msg package):
1 - Go to /home/daniel/tello_ws/src/name_pkg
2 - Open CMakeLists.txt and package.xml, then change all "std_msg" string to "std_msgs"
3 - Save changes in files and execute "catkin_make" in VS Terminal again, it should compile n

Open new linux terminal and run:
roscore   ## Execute master node

Open new linux terminal and run:
roslaunch tello_driver tello_node.launch    ## Execute tello driver node

Open new linux terminal and run:
rosrun image_viewer_tello image_viewer.py  ## Start image viewer node
```

6.10 QR Detection with Tello Camera

https://alfredo-reyes-montero.gitbook.io/tello-dji/frameworks/ros http://wiki.ros.org/cv_bridge https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImage https://github.com/Eric-Canas/qrdet https://github.com/Eric-Canas/qreader

```
Open new linux terminal and run:
pip3 install qrdet     #### QR Detector based in YOLOv8
pip install qreader    #### Alternate detector (QR detector + QR decoder)
sudo apt-get install libzbar0   #### For alternate detector (Required for qreader)

Open new linux terminal and run:
cd tello_ws/src
catkin_create_pkg qr_detector_tello cv_bridge sensor_msgs image_transport roscpp ros
## Create package with dependencies to use

In VS Code:
Open working folder (use __File -> Open Folder__ option) "tello_ws":
In tello_ws/src/qr_detector_tello/src:
New File (qr_detector.py)
Write or copy node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/qr_detector_tello/src ## Go to package src folder
chmod +x qr_detector.py ## Give execution privileges to node (qr_detector.py)
cd  ## Go to home
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project

Open new linux terminal and run:
roscore   ## Execute master node
roslaunch tello_driver tello_node.launch    ## Execute tello driver node
rosrun qr_detector_tello qr_detector.py  ## Start qr_detector node
```

6.11 Install Modified Driver

https://github.com/JoseBalbuena181096/catkin_ws_ROS_TELLO

* Clone repo from link above.
* Unzip downloaded file
* Go to src folder in unzipped repo and copy jose_driver_tello folder to tello_ws/src
For this work, the folder was renamed to driver_tello_mod and
the python file was renamed to tello_driver_mod.py.
Due to the change of name of this drive, also it is necessary to change the name
in CMakeLists.txt and package.xml files.

Go to workspace folder (tello_ws) and open a new linux terminal:
Run: catkin_make

Open new linux terminal and run:
roscore   ## Execute master node
rosrun driver_tello_mod tello_driver_mod.py   ## Execute tello driver node

Open new linux terminal and run:
rosrun rqt_image_view rqt_image_view /tello/image_raw ## Start camera viewer

6.12 QR Follower-Lander for Tello

https://alfredo-reyes-montero.gitbook.io/tello-dji/frameworks/ros http://wiki.
ros.org/cv_bridge https://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCV
https://github.com/Eric-Canas/qrdet

Open new linux terminal and run:
cd tello_ws/src
catkin_create_pkg follow_qr_tello cv_bridge geometry_msgs sensor_msgs image_transport roscpp
Create package with dependencies to use

In VS Code:
Open working folder (use __File -> Open Folder__ option) "tello_ws":
In tello_ws/src/follow_qr_tello/src:

```
New File (follow_qr_tello.py)
Write or copy node code
Save
In Terminal Tab -> New Terminal:
cd tello_ws/src/follow_qr_tello/src ## Go to package src folder
chmod +x follow_qr_tello.py ## Give execution privileges to node (follow_qr_tello.py
cd  ## Go to home
cd tello_ws ## Go to home of workspace
catkin_make  ## Compile project

Open new linux terminal and run:
roscore   ## Execute master node
rosrun driver_tello_mod tello_driver_mod.py     ## Execute tello driver node mod
roslaunch follow_qr_tello follow_qr_tello.launch image_reduction:=60  ## Start qr_fo
rosrun qr_detector_tello qr_detector.py  ## Start qr_detector node
```

Bibliography

[1] I. González-Hernández, S. Salazar, R. Lozano, and O. Ramírez-Ayala, "Real-time improvement of a trajectory-tracking control based on super-twisting algorithm for a quadrotor aircraft," *Drones*, vol. 6, no. 2, p. 36, 2022.

[2] L. Joseph, *Robot operating system (ros) for absolute beginners*. Springer, 2018.

[3] M. A. Wani, F. A. Bhat, S. Afzal, and A. I. Khan, *Advances in deep learning*. Springer, 2020.

[4] V. Zocca, G. Spacagna, D. Slater, and P. Roelants, *Python Deep Learning*. Packt Publishing Ltd, 2017.

[5] N. K. Manaswi, N. K. Manaswi, and S. John, *Deep learning with applications using python*. Springer, 2018.

[6] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[7] RangeKing, "Brief summary of yolov8 model structure," https://github.com/ultralytics/ultralytics/issues/189, 2024, accessed: 2024-06-10.

[8] G. C. Goodwin, S. F. Graebe, M. E. Salgado *et al.*, *Control system design*. Prentice Hall Upper Saddle River, 2001, vol. 240.

[9] L. Yu, C. Luo, X. Yu, X. Jiang, E. Yang, C. Luo, and P. Ren, "Deep learning for vision-based micro aerial vehicle autonomous landing," *International Journal of Micro Air Vehicles*, vol. 10, pp. 171–185, 6 2018, rEAD THIS.

[10] S. Lin, L. Jin, and Z. Chen, "Real-time monocular vision system for uav autonomous landing in outdoor low-illumination environments," *Sensors*, vol. 21, 9 2021.

[11] P. H. Nguyen, M. Arsalan, J. H. Koo, R. A. Naqvi, N. Q. Truong, and K. R. Park, "Lightdenseyolo: A fast and accurate marker tracker for autonomous uav landing by visible light camera sensor on drone," *Sensors*, vol. 18, no. 6, p. 1703, 2018.

[12] A. A. Cabrera-Ponce and J. Martínez-Carranza, "Onboard cnn-based processing for target detection and autonomous landing for mavs," in *Mexican Conference on Pattern Recognition*. Springer, 2020, pp. 195–208.

[13] L. Wu, C. Wang, P. Zhang, and C. Wei, "Deep reinforcement learning with corrective feedback for autonomous uav landing on a mobile platform," *Drones*, vol. 6, no. 9, p. 238, 2022.

[14] M. Piponidis, P. Aristodemou, and T. Theocharides, "Towards a fully autonomous uav controller for moving platform detection and landing," in *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*. IEEE, 2022, pp. 180–185.

[15] Y. Lu, Z. Xue, G.-S. Xia, and L. Zhang, "A survey on vision-based uav navigation," *Geo-spatial information science*, vol. 21, no. 1, pp. 21–32, 2018.

[16] R. G. L. Narayanan and O. C. Ibe, "Joint network for disaster relief and search and rescue network operations," in *Wireless Public Safety Networks 1*. Elsevier, 2015, pp. 163–193.

[17] J. Hall and K. Mohseni, *Micro Aerial Vehicles*. Boston, MA: Springer US, 2013, pp. 1–10. [Online]. Available: https://doi.org/10.1007/978-3-642-27758-0_892-2

[18] M. Idrissi, M. Salami, and F. Annaz, "A review of quadrotor unmanned aerial vehicles: Applications, architectural design and control algorithms," *Journal of Intelligent & Robotic Systems*, vol. 104, no. 2, pp. 1–33, 2022.

[19] S. de Comunicaciones y Transportes, "Norma oficial mexicana nom-107-sct3-2019," *Diario Oficial de la Federación*, 2019.

[20] C. Bernardeschi, A. Fagiolini, M. Palmieri, G. Scrima, and F. Sofia, "Ros/gazebo based simulation of co-operative uavs," in *International Conference on Modelling and Simulation for Autonomous Systems*. Springer, 2018, pp. 321–334.

[21] H. A. Romo, J. C. Realpe, and P. E. Jojoa, "Análisis de señales emg superficiales y su aplicación en control de prótesis de mano," *Revista Avances en Sistemas e Informática*, vol. 4, no. 1, pp. 127–136, 2007.

[22] J. Loy, *Neural Network Projects with Python: The ultimate guide to using Python to explore the true power of neural networks through six projects*. Packt Publishing Ltd, 2019.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[24] P. Soviany and R. T. Ionescu, "Optimizing the trade-off between single-stage and two-stage deep object detectors using image difficulty prediction," in *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2018, pp. 209–214.

[25] Y. Xiao, Z. Tian, J. Yu, Y. Zhang, S. Liu, S. Du, and X. Lan, "A review of object detection based on deep learning," *Multimedia Tools and Applications*, vol. 79, pp. 23 729–23 791, 2020.

[26] P. Adarsh, P. Rathi, and M. Kumar, "Yolo v3-tiny: Object detection and recognition using one stage improved model," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694.

[27] M. Hussain, "Yolov1 to v8: Unveiling each variant–a comprehensive review of yolo," *IEEE Access*, vol. 12, pp. 42 816–42 833, 2024.

[28] Y.-G. Wang and H.-H. Shao, "Optimal tuning for pi controller," *Automatica*, vol. 36, no. 1, pp. 147–152, 2000.

[29] K. Bhujbal and S. Barahate, "Custom object detection based on regional convolutional neural network & yolov3 with dji tello programmable drone," in *7th International Conference on Innovation & Research in Technology & Engineering (ICIRTE)*, 2022.

[30] J. Ángel Balbuena Palma, "Ros-tello driver," https://github.com/JoseBalbuena181096/catkin_ws_ROS_TELLO, accessed: January 2024.

[31] E. Cañas, "Qrdet," September 2023, accessed: January 2024. [Online]. Available: https://github.com/Eric-Canas/qrdet

[32] V. Jayaswal, "Performance metrics: Confusion matrix, precision, recall, and f1 score," https://towardsdatascience.com/performance-metrics-confusion-matrix-precision-recall-and-f1-score-a8fe076a2262, 2020.

[33] M. Buckland and F. Gey, "The relationship between recall and precision," *Journal of the American society for information science*, vol. 45, no. 1, pp. 12–19, 1994.