



الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic Republic of Algeria

وزارة التعليم العالي والبحث العلمي

Ministry of Higher Education and Scientific Research

المدرسة الوطنية العليا للتكنولوجيا المتقدمة

Ecole Nationale Supérieure des Technologies Avancées



GÉNIE ELECTRIQUE ET INFORMATIQUE INDUSTRIEL

Final Year Project to Obtain the Diploma of
Engineering

- Field -

Telecommunication

- Specialty -

Telecommunication Systems and Networking

- Subject -

**Forest Fire Prediction and Detection
System by Drone Equipped with AI**

Realized by

Khaled Gasmi & Yasser Cherfaoui

Members of the Jury :

Mme Khadidja Zellat	Chair
Mme Djamila Bendouda	Examiner
Mme Nafissa Rezki	Examiner
Mr El Hadi Khoumeri	Supervisor

Algiers, Jun 25th 2024

Academic year 2023-2024

Dedication

Khaled Gasm

My father, whose unwavering support fueled my determination throughout this journey.

My mother, whose spiritual encouragement provided unwavering strength.

My siblings, for their constant love and understanding.

To all my friends, especially those closest to me, for their unwavering camaraderie and belief in my abilities.

Dr. El Hadi Khoumeri, our supervisor, whose guidance and expertise were instrumental in shaping this project.

To all those dear to me and to everyone who, near or far, offered their support in countless ways.

Yasser Cherfaoui

To my beloved parents, whose unwavering support and encouragement have been my greatest strength throughout this journey. Their love and sacrifices have laid the foundation for all my achievements.

To my esteemed mentor, Mr. El-Hadi Khoumeri, I extend my deepest gratitude. Your guidance, wisdom, and unwavering belief in my abilities have been invaluable.

To my dear friends, Ilyas Brahmi and Mohamed Amine Djaballah, thank you for your constant companionship, support, and encouragement. Your friendship has been a source of great motivation and joy.

Acknowledgment

We would like to express our deepest gratitude to everyone who has supported us throughout the development of this thesis. This work would not have been possible without the guidance, encouragement, and assistance of many individuals.

First and foremost, we would like to thank our parents for their unwavering support and love. Their trust in us has been a constant source of motivation and strength, allowing us to pursue our goals with confidence and determination. We are forever grateful for their sacrifices and belief in our abilities.

We are immensely grateful to our supervisor, Mr. El Hadi Khoumeri, for their invaluable guidance, insightful advice, and continuous encouragement throughout this project. Their expertise and dedication have been instrumental in shaping this work, and their constructive feedback has significantly improved the quality of this thesis. Thank you for being a constant source of inspiration and for believing in our vision.

A special thank you goes to our friends, who have stood by us since the beginning of this journey. Your support, encouragement, and understanding have been vital in helping us overcome the challenges we faced. We are deeply thankful for your companionship and for always being there to share both the highs and lows of this endeavor.

This thesis presents the development of a user interface to monitor drone states and notify users of smoke, fire, or human presence in forests. It is the culmination of hard work, perseverance, and the collective support of those mentioned above. We are sincerely grateful to each one of you for your contributions to this project.

ملخص

تقدم هذه الأطروحة نهجاً مبتكراً للكشف عن حرائق الغابات باستخدام الطائرات بدون طيار المجهزة بتقنيات الذكاء الاصطناعي. من خلال الاستفادة من طائرة تيلو دي جي آي، يدمج النظام رؤية الكمبيوتر والتعلم المعزز للكشف عن الحرائق في الوقت الفعلي. حقق المشروع أداءً جيداً خلال الاختبارات، مما يدل على أداء موثوق. تسهم هذه الأبحاث في تحسين استراتيجيات إدارة الكوارث، وتقدم حلاً قابلاً للتطوير للكشف المبكر عن حرائق الغابات.

الكلمات المفتاحية: كشف حرائق الغابات، الطائرات بدون طيار، الذكاء الاصطناعي، رؤية الكمبيوتر، التعلم المعزز، تيلو دي جي آي.

Abstract

This thesis presents an innovative approach to wildfire detection using UAVs equipped with AI technologies. Leveraging the DJI Tello drone, the system integrates computer vision and reinforcement learning for real-time fire detection. The project achieved significant success, demonstrating reliable performance. This research contributes to enhancing disaster management strategies, offering a scalable solution for early wildfire detection.

Keywords: Wildfire detection, UAV, AI, Computer vision, Reinforcement learning, DJI Tello.

Résumé

Cette thèse présente une approche innovante de la détection des incendies de forêt en utilisant des UAVs équipés de technologies d'intelligence artificielle. En tirant parti du drone DJI Tello, le système intègre la vision par ordinateur et reinforcement learning pour la détection des incendies en temps réel. Le projet a rencontré un succès significatif, démontrant des performances fiables. Cette recherche contribue à l'amélioration des stratégies de gestion des catastrophes, offrant une solution évolutive pour la détection précoce des incendies de forêt.

Mots-clés : Détection des incendies de forêt, UAV, IA, Vision par ordinateur, Reinforcement learning, DJI Tello.

Contents

List of Figures	i
List of Tables	iii
Acronyms	v
Introduction	1
1 Theoretical Framework and Literature Review	3
1.1 Introduction	3
1.2 Using Reinforcement Learning for Auto-Patrolling Drones	3
1.2.1 Environment	4
1.2.2 Reinforcement Learning Framework	4
1.2.3 Training the Drone	5
1.3 Computer vision for fire detection	5
1.3.1 Object detection	5
1.3.2 Object detectors	6
1.3.3 Requirement and objectives	8
1.3.4 Why YOLO	9
1.3.5 Basics of YOLO algorithm	12
1.4 Development Theory: Implementing a Dashboard with Flutter	14
1.4.1 Technologies Used	15
1.5 Theoretical Framework: Simulating Drone Operations with the Godot Game Engine	16
1.5.1 ForestWings	16
1.6 State of the art	17
2 Hardware and Software	20
2.1 Introduction	20
2.2 Drone Tello	20

2.2.1	Technical characteristics	20
2.2.2	Tello SDK 2.0	21
2.3	Torch	22
2.3.1	Torch's Role in Reinforcement Learning	22
2.3.2	Key Features of Torch Useful for The Project	22
2.3.3	Advantages of Using Torch	23
2.4	OpenCV	23
2.4.1	Video and Image Processing Capabilities	23
2.4.2	challenges and solutions	24
2.4.3	Advantages of Using OpenCV	24
2.5	YOLO	25
2.6	Flutter	25
2.6.1	Key Features of Flutter	25
2.6.2	Architecture	26
2.7	GoDot game engine	26
2.7.1	Key Features of Godot	26
2.7.2	Architecture	27
3	Methodology/Project Work	29
3.1	Introduction	29
3.2	System Architecture	30
3.3	Computer Vision System	31
3.3.1	Computer Vision Algorithms Used for Wildfire Detection	31
3.3.2	Data Preprocessing and Model Training	32
3.3.3	Performance Metrics and Accuracy of the Detection System	39
3.4	Autonomous Navigation and Control	43
3.4.1	Path Planning Algorithms	44
3.4.2	Real-time Decision-making for Obstacle Avoidance	46
3.4.3	Methods for Ensuring Efficient Coverage of the Patrol Area	48
3.5	Drone Communication and Video Stream	48
3.5.1	Communication Protocol	49
3.6	Calculation of Fire Ignition Risk	51
3.6.1	Fire Risk Indices	52
3.6.2	Simple Fire Danger Index	52
3.6.3	Implementation	53
3.6.4	Fire Danger Levels	53
3.7	Dashboard Interface and Visualization	54

3.8	Project Integration	54
3.8.1	Integrating the Weather API into the Dashboard	54
3.8.2	Integrating the Model and Alarm Launching	59
3.8.3	Integrating the auto patrolling algorithm into the simulator	62
3.8.4	Integrating the map of the region in the dashboard	63
3.9	Future Enhancements and Scalability	64
3.9.1	Potential Improvements to the System Design	64
3.9.2	Plans for Scaling the System to Cover Larger Areas or Different Types of Environments	65
3.9.3	Considerations for Future Technological Advancements	65
4	Test and Validation	67
4.1	Introduction	67
4.2	Drone Patrolling Testing and Validation	67
4.2.1	Testing Methodology	67
4.2.2	Validation Results	68
4.2.3	Discussion	69
4.3	Computer Vision Model Test and Validation	69
4.3.1	Validation Through the Validation Dataset Images	70
4.3.2	Pushing the Limits Using Extreme Images	72
4.3.3	Testing Through Real Wild Fire Videos	73
4.4	Simulator Test and Validation	74
4.4.1	Validation Methodology	74
4.4.2	Validation Results	75
4.4.3	Discussion	75
4.5	Integrated System Validation	76
4.5.1	Fire Ignition Risk and Patrolling Frequency Allocation	76
4.5.2	Drone Communication	77
4.5.3	Capture the Stream from the Drone	77
	Conclusion	83
	Bibliography	84
	A Important functions and methods used in building the project	A

List of Figures

1.1	Stages of object detector	6
1.2	Main component of Object detector	7
1.3	Component of Object detector	8
1.4	Average Precision (AP) of YOLO on COCO Benchmark	10
1.5	by Frames per Second (FPS) of YOLO on COCO Benchmark	10
1.6	Performance of different YOLO vesions on COCO Object detection dataset	11
1.7	The parameters of object for localisation in yolo algorithm	12
1.8	Illustration of grid image	13
1.9	How result of the yolo model look like	14
1.10	3D Model of the used Tello drone.	16
1.11	Dirt skin used in our environment.	16
1.12	3D Model of Tree.	17
1.13	Terrain texture skin.	17
2.1	Drone Tello EDU	21
3.1	project diagram	30
3.2	Image labeled in YOLO format	32
3.3	Folder structure of the dataset.	35
3.4	Training and Validation Losses for Box, Classification, and DFL, along with Precision, Recall, and mAP metrics.	40
3.5	Confusion Matrix with Raw Values.	41
3.6	Normalized Confusion Matrix.	41
3.7	Confusion Matrix with Normalized and Raw Values.	41
3.8	Precision-Confidence Curve for Fire, Smoke, and All Classes.	42
3.9	Recall-Confidence Curve for Fire, Smoke, and All Classes.	42
3.10	F1-Confidence Curve for Fire, Smoke, and All Classes.	42
3.11	precision and recall confidence curve	42

3.12 Updated Training and Validation Losses for Box, Classification, and DFL, along with Precision, Recall, and mAP metrics.	43
3.13 Updated Normalized Confusion Matrix.	44
3.14 Updated Precision-Confidence Curve for Fire, Smoke, and All Classes.	45
3.15 Updated Recall-Confidence Curve for Fire, Smoke, and All Classes.	45
3.16 Updated F1-Confidence Curve for Fire, Smoke, and All Classes.	45
3.17 precision, recall and f1 confidence curve after update	45
3.18 Exploration patrolling	46
3.19 Exploration Matrix	46
3.20 Dashboard Interface	55
3.21 illustrative image of swarm drones patrolled a region.	66
4.1 Labeled validation batch	71
4.2 Predicted validation batch	71
4.3 Validation image 1	78
4.4 Validation image 2	78
4.5 Validation image 3	78
4.6 Validation image 4	78
4.7 Validation image 5	78
4.8 Validation image 6	78
4.9 Validation image 7	78
4.10 Validation image 8	78
4.11 Validation image 9	78
4.12 Validation image 1	79
4.13 Validation image 2	79
4.14 Validation image 3	79
4.15 Validation image 4	79
4.16 Validation image 5	79
4.17 Validation image 6	79
4.18 ForestWings Simulator running at 100 FPS	79
4.19 Visualization of the currant risk and time to launch the drone	79
4.20 Graph of the risk in function of days	80
4.21 States of Tello	81
4.22 Interface to send Commands to Tello	81
4.23 Capture the Drone's Stream	82

List of Tables

- 2.1 Technical characteristics of Tello 28
- 3.1 Simple Fire Danger Index potential scale. 53
- 4.1 Model validation results 71
- 4.2 Person detection model validation results 71
- 4.3 Risk level predictions based on meteorological data 76

List of Algorithms

1	Python function to permit the classes label of a model.	33
2	Python function to copy text files from a source folder to a destination. . .	34
3	Dynamic Window Approach (DWA)	47
4	Reinforcement Learning for Obstacle Avoidance	48
5	Python function to send a list of command to the drone.	49
6	Python function to capture video stream.	50
7	Model integration with the stream.	51
8	Weather Response Class Definition	55
9	Weather Service Class Definition	56
10	onReady Function Definition	58
11	WeatherController Class Definition	58
12	Flask Application Example	59
13	Fire Alarms Class	60
14	SnackBar Global Schema	61
15	Setting Up <code>ever()</code>	62
16	Validation Using Ultralytics YOLO	70
17	Code to apply the model on a group of images contained in a folder and save the results.	72
18	xml fire for an image annotation.	A
19	Code to convert xml based format to yolo format.	B
20	Location field class	C
21	Current field class	D

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

CNN Convolutional Neural Network.

DBDI Modified Drought Index.

DFL Distributed Focus Loss.

DWA Dynamic Window Approach.

FMI Fuel Moisture Index.

FPS Frame Per Second.

FWI Fire Weather Index.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

IDE Integrated Development Environment.

KBDI Keetch-Byram Drought Index.

mAP mean Average Precision.

R-CNN Region-based Convolutional Neural Network.

RL Reinforcement Learning.

SDK Software Development Kit.

SSD Single Shot Detector.

UAV Unmanned Aerial Vehicle.

VGG Visual Geometry Group.

YOLO You Only Look Once.

Introduction

Wildfires have emerged as a significant global concern, exacerbated by climate change and human activities. These fires pose severe threats to ecosystems, human life, and property, necessitating rapid detection and effective management strategies to mitigate their devastating impacts. The early detection and prompt response to wildfires are crucial in preventing widespread damage and loss. However, traditional methods of monitoring and detecting wildfires often fall short due to limitations in coverage, speed, and accuracy.

To combat this escalating problem, integrating advanced technologies into wildfire detection and prevention systems has become imperative. Unmanned Aerial Vehicles (UAVs), equipped with cutting-edge sensors and artificial intelligence (AI), present a promising solution for enhancing the efficiency and effectiveness of wildfire management. UAVs offer unparalleled advantages, including real-time monitoring, high-resolution imagery, and the ability to access remote and hazardous areas without risking human lives.

In recent years, advancements in UAV technology have significantly improved their capabilities, making them invaluable tools in various fields, including disaster response and environmental monitoring. This thesis proposes an innovative approach to wildfire detection using a DJI Tello drone equipped with advanced AI technologies. The primary objective is to develop an automated system capable of monitoring forest areas and quickly detecting the onset of fires, thereby facilitating rapid intervention and preventing environmental disasters.

The proposed solution involves leveraging the capabilities of the DJI Tello drone in conjunction with state-of-the-art AI algorithms for real-time wildfire detection. By integrating computer vision techniques and reinforcement learning, the system aims to enhance the drone's ability to autonomously patrol forest areas, identify potential fire hazards, and promptly alert authorities for swift action. This approach not only improves the accuracy and speed of wildfire detection but also reduces the dependency on manual surveillance methods.

The thesis is structured as follows:

- **Theoretical Framework and Literature Review:** This chapter provides a comprehensive overview of the theoretical foundations essential for developing UAV-based systems for forest patrolling and fire detection. It examines reinforcement learning, computer vision, and the use of the Godot game engine for UAV simulation.
- **Hardware and Software:** An in-depth discussion of the hardware and software components used in the project, including the DJI Tello drone, Torch, OpenCV, YOLO, Flutter, and the Godot game engine. This chapter highlights the technical specifications and functionalities that enable the proposed solution.
- **Methodology/Project Work:** This chapter outlines the system architecture, computer vision algorithms, autonomous navigation and control mechanisms, drone communication protocols, and the integration of these components into a cohesive system. It also discusses the methods for calculating fire ignition risk and the development of the dashboard interface.
- **Test and Validation:** A detailed account of the testing and validation processes undertaken to assess the performance of the proposed system. This chapter includes the validation results of the drone's patrolling capabilities, the accuracy of the computer vision model, and the effectiveness of the integrated system in real-world scenarios.

Chapter 1

Theoretical Framework and Literature Review

1.1 Introduction

Unmanned aerial vehicles (UAVs) have transformed fields such as surveillance, disaster response, and environmental conservation. In wildfire detection and prevention, UAVs with advanced technologies enhance traditional forest management. This chapter presents the theoretical frameworks essential for developing UAV-based systems for forest patrolling and fire detection.

We first examine reinforcement learning, which enables UAVs to autonomously optimize patrolling strategies using techniques like Q-learning and deep reinforcement learning [1]. Next, we discuss the role of computer vision in real-time wildfire detection, covering algorithms for fire, smoke and person detection [2].

Additionally, we highlight the use of the Godot game engine for developing UAV simulators. These simulators allow for the testing and refinement of UAV algorithms in virtual environments that mimic real-world conditions. Finally, a review of recent advancements in these technologies sets the foundation for the development of an innovative UAV-based system for effective forest patrolling and wildfire detection.

1.2 Using Reinforcement Learning for Auto-Patrolling Drones

The project aims to develop a "launch-and-forget" drone capable of autonomous patrolling using Reinforcement Learning (RL). RL is a type of machine learning where an agent

learns to make decisions by performing actions in an environment to maximize cumulative reward. This approach is particularly well-suited for developing autonomous systems like drones, where the goal is to learn optimal policies for complex tasks through interaction with the environment.

1.2.1 Environment

- **State Space:** The state space includes all possible states the drone can be in, such as its position, velocity, orientation, battery level, and sensory inputs.
- **Action Space:** The action space comprises all possible actions the drone can take, including changes in pitch, roll, yaw, and thrust, as well as higher-level actions like moving to specific waypoints.

1.2.2 Reinforcement Learning Framework

1.2.2.1 Agent and Environment Interaction

- **Agent:** The drone, which makes decisions based on its policy.
- **Environment:** The physical world or a simulated environment in which the drone operates.

1.2.2.2 Markov Decision Process (MDP)

- **States (S):** A set of all possible states (e.g., GPS coordinates, velocity vectors, obstacle positions).
- **Actions (A):** A set of all possible actions (e.g., throttle adjustments, direction changes).
- **Transition Function (T):** Probability of moving from one state to another given an action.
- **Reward Function (R):** Immediate reward received after transitioning from one state to another due to an action.

1.2.2.3 Learning Algorithm

- **Q-Learning:** A value-based RL algorithm where the agent learns the value of each action in each state.

- **Deep Q-Network (DQN):** Uses neural networks to approximate Q-values for handling large state and action spaces.
- **Policy Gradient Methods:** Learn a parameterized policy directly, using methods like Proximal Policy Optimization (PPO) or Advantage Actor-Critic (A2C).

1.2.3 Training the Drone

1.2.3.1 Simulation Environment

- **Simulator:** Use ForestWings¹ to create a virtual environment for initial training.
- **Scenarios:** Define various patrolling scenarios, including obstacle-rich environments, different weather conditions, and varying terrains.

1.2.3.2 Reward Shaping

- **Basic Rewards:** Positive rewards for reaching waypoints and completing patrols; negative rewards for collisions, straying off course, or inefficient paths.
- **Dense vs. Sparse Rewards:** Balance between giving frequent small rewards and occasional large rewards to guide the learning process.

1.3 Computer vision for fire detection

In this section, we delve into the approach of object detection, with a specific focus on the principles underlying the YOLO (You Only Look Once) algorithm [3]. We will explore the fundamental workings of a YOLO object detector, from its reliance on labeled data to its unique real-time detection capabilities. Additionally, we'll elucidate the rationale behind our choice of YOLO for our project, highlighting its efficiency and suitability for our objectives. Through this exploration, we aim to provide a comprehensive understanding of the YOLO algorithm and its significance in the context of our project's goals.

1.3.1 Object detection

Object detection is a pivotal task in computer vision aimed at identifying and locating instances of specific objects within digital images or video frames. In the context of wildfire detection projects, object detection involves the automatic recognition of crucial elements

¹ForestWings, is a software developed by our team to make tests on the patrolling algorithm.

related to wildfires, such as flames and smoke, within images or video feeds captured by surveillance cameras or drones. The primary objective is to choose sophisticated computational models capable of accurately pinpointing the presence and location of these wildfire-related objects in real-time or near-real-time scenarios. This facilitates prompt and effective response strategies, aiding in early detection and mitigation of wildfires to minimise their adverse impacts on the environment and human lives.

1.3.2 Object detectors

Object detectors rely heavily on the utilisation of advanced deep learning architectures, notably convolutional neural networks (CNNs). Through extensive training on large-scale annotated datasets, these models acquire the capability to discern intricate patterns and distinctive features indicative of different objects within visual data. The integration of object detection methodologies equips machines with the capacity to interpret and analyse the visual environment, facilitating profound advancements across diverse domains. This technological paradigm enables a wide array of applications, ranging from autonomous vehicles and surveillance systems to medical imaging and industrial automation, thereby catalysing transformative progress in computational perception and decision-making. To elucidate the intricate operational mechanisms of an object detector, let's deconstruct its functionality into distinct stages.

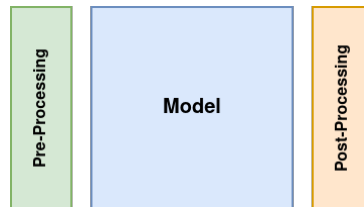


Figure 1.1: Stages of object detector

In the object detector depicted above, three key components drive its functionality. Firstly, preprocessing optimizes input data by resizing images, cleaning noise, normalising characteristics, and annotating object bounding boxes. These preparatory steps are essential for refining raw data and ensuring compatibility with the model, facilitating seamless functionality and accurate object detection.

In the post-processing phase, detected objects undergo refinement to enhance accuracy and usability. This involves filtering out redundant detections, merging overlapping bounding boxes, and classifying objects based on their attributes. These steps ensure that the final output is optimized for further analysis and decision-making.

now let's get back to the main component which is the model, which itself contain other component

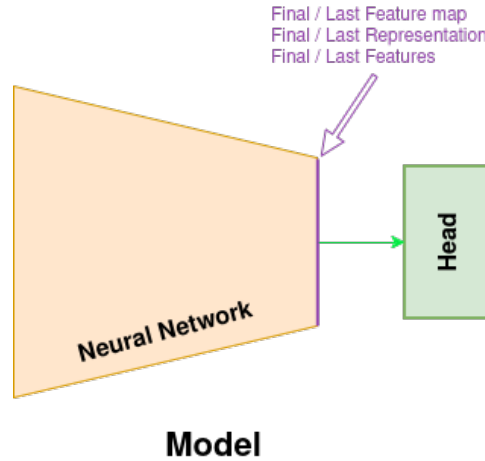


Figure 1.2: Main component of Object detector

In the current state of the art [4], our model comprises a neural network, specifically a convolutional neural network (CNN), tailored for image processing tasks. Notable CNN architectures include ResNet [5], VGG, and Inception, each contributing to the model's capabilities. Within this network, multiple layers perform intricate operations, yet our focus lies primarily on the final layer known as the "last features layer." This layer serves as input for another neural network responsible for object detection, aptly named the "detection head." Meanwhile, the CNN responsible for extracting these features is termed the "backbone." However, relying solely on the last features layer may overlook valuable information. To optimise performance, we explore leveraging the diverse layers within the backbone. By incorporating features from various depths, we enrich our understanding of the input image. To achieve this, we introduce an intermediary neural network, the "Neck," positioned between the backbone and the detection heads. Here, refinement occurs through the amalgamation and sampling of feature maps. The resulting output, termed the "refined feature map," is subsequently fed into the respective detection heads. This layered architecture enhances the model's ability to discern intricate details and improve overall detection accuracy, as illustrated in the diagram below 1.3.

In neural networks, a fascinating phenomenon emerges: the deeper the network, the more abstract the information it processes, accompanied by an expansion in the receptive field [6]. Consequently, feature maps closer to the input layer excel in discerning finer details, rendering them adept at detecting smaller objects. Conversely, those situated farther into the network demonstrate heightened efficacy in identifying larger objects, owing to their broader receptive fields.

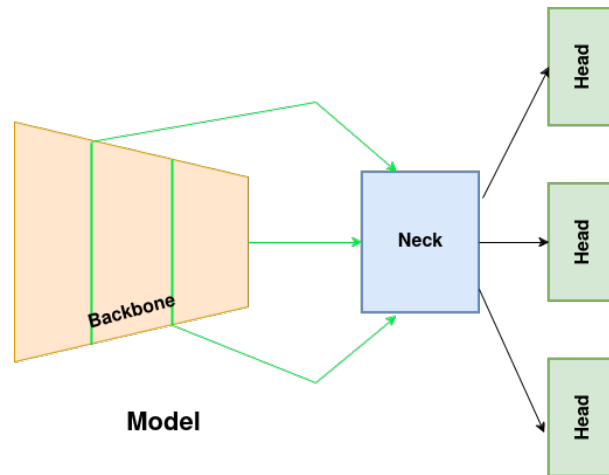


Figure 1.3: Component of Object detector

These are the fundamental components of an object detector, and there are several existing architectures available. As we embark on developing our own, we'll leverage an existing architecture. The question then becomes: which architecture to select? Let's explore some of the most renowned options:

- YOLO (You Only Look Once)
- SSD (Single Shot Detector)
- Mask R-CNN (Mask Region-Based Convolutional Neural Network)
- Faster R-CNN (Faster Region-Based Convolutional Neural Network)
- EfficientDet
- RetinaNet

To make an informed decision, we must first assess our specific requirements and objectives to determine the most appropriate choice.

1.3.3 Requirement and objectives

In our project, we aim to deploy a computer vision model for wildfire detection using drones. Given the dynamic nature of wildfire incidents and the need for real-time response, our solution must meet specific requirements to effectively address this critical task. Firstly, considering the limited computational resources available, we need a model architecture that strikes a balance between accuracy and efficiency. This architecture should support fine-tuning with new data collected during drone patrols, allowing us

to continuously update and improve the model's performance over time without overwhelming the onboard hardware. Furthermore, robustness to environmental conditions is paramount, as drones may encounter various challenges such as smoke, haze, and changing lighting conditions during patrols. Hence, the selected architecture must exhibit resilience to such variability and maintain reliable performance under adverse conditions. Additionally, given the remote and often inaccessible locations where wildfires occur, our solution should be capable of real-time inference to enable timely detection and response. Lastly, the model must be deployable on drones with minimal computational overhead, ensuring efficient utilization of onboard resources and enabling autonomous operation without the need for constant human intervention. By addressing these requirements, our project aims to develop a reliable and efficient system for wildfire detection using drones.

1.3.4 Why YOLO

1.3.4.1 Compare some object detection algorithms

In the domain of real-time wildfire detection initiatives, the crucial need for precise and rapid hazard identification is undeniable. To assess the efficacy of different detection algorithms, the Microsoft COCO dataset [7] serves as a renowned standard for evaluation. This dataset subjects models to rigorous scrutiny, utilising the Mean Average Precision (MAP) metric to offer a thorough evaluation of their capabilities across various scenarios. Our exploration aims to provide an initial overview of leading real-time object detection algorithms tailored to the specific requirements of our wildfire detection project.

1.3.4.2 The Best Real-Time Object Detection Algorithm for our need

Here is a graphical representation the performance of different algorithms from vision.io [8]:

The line graph 1.4 delineates the performance trajectories of various object detection models over a two-year period, underscoring the evolution of object detection methodologies. Notably, a specific variant of the YOLO algorithm, version of "YOLOV7-E6E (36 fps)," emerges as a standout performer, boasting both high accuracy and a rapid frame rate of 36 frames per second (fps). This distinction positions the YOLO algorithm as particularly well-suited for real-time applications, where timely detection and response are paramount. By achieving superior accuracy while maintaining efficient processing speeds, YOLO offers a compelling advantage over other object detection models, ensuring swift and accurate identification of objects in dynamic environments. This emphasizes the significance of YOLO as a pivotal tool in real-time applications, where its blend of

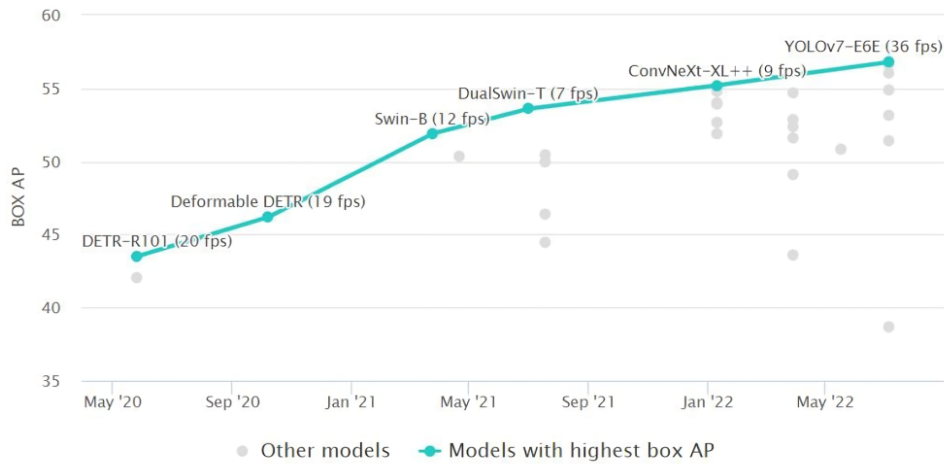


Figure 1.4: Real-time Object Detection on COCO Benchmark: The state-of-the-art by Average Precision (AP)

accuracy and efficiency elevates performance and efficacy. and in terms of Inference Time, we analyze the graph in the figure below 1.5 [7]:

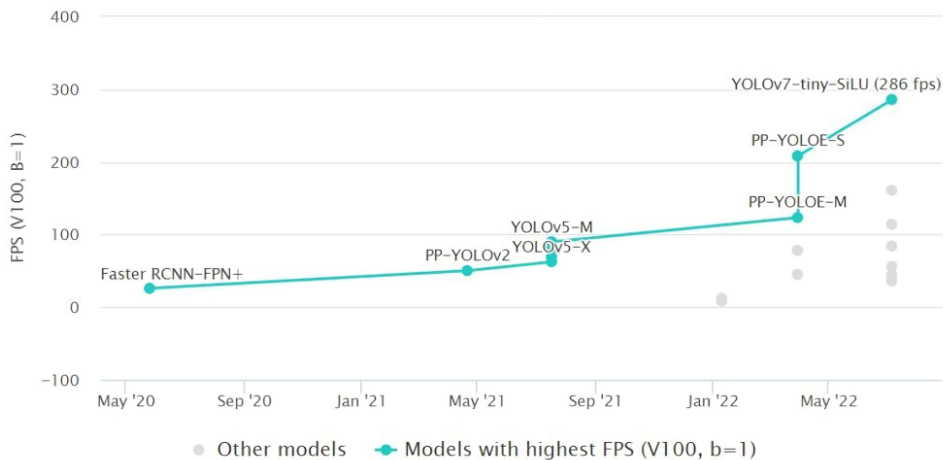


Figure 1.5: The state-of-the-art by Frames per Second (FPS): The leading computer vision algorithm for real-time object detection on COCO can process 286 frames per second (YOLOv7), and is faster than YOLOv5, YOLOv4, YOLOR, and YOLOv3.

It's evident that YOLO stands out from other object detection models in terms of processing speed. The graph showcases the performance trajectories of various algorithms over time, and while many achieve high accuracy, YOLO, particularly the "YOLOV7-E6E" variant, excels in frame rate. Unlike some detection algorithms that require multiple passes through an image to identify objects, YOLO leverages a single network, significantly reducing processing time. This streamlined approach, reflected in the model's

high FPS (36 fps) in the graph, translates to faster object identification in dynamic environments. This speed advantage, coupled with YOLO's maintained accuracy, makes it a compelling choice for real-time applications where timely detection and response are crucial.

1.3.4.3 Latest of YOLO

When examining the latest advancements in the YOLO (You Only Look Once) algorithm, a new research paper titled "YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information" [9] was unveiled in March 2024. This paper introduces significant modifications to certain components of the YOLO object detectors, particularly focusing on the neck and backbone structures, while retaining the head architecture from YOLOv3. These alterations aim to enhance the algorithm's performance and adaptability to diverse tasks. As a testament to its continuous evolution, the official GitHub repository of YOLOv9 features a graph illustrating the algorithm's performance across various releases in terms of Average Precision (AP) relative to the number of parameters.

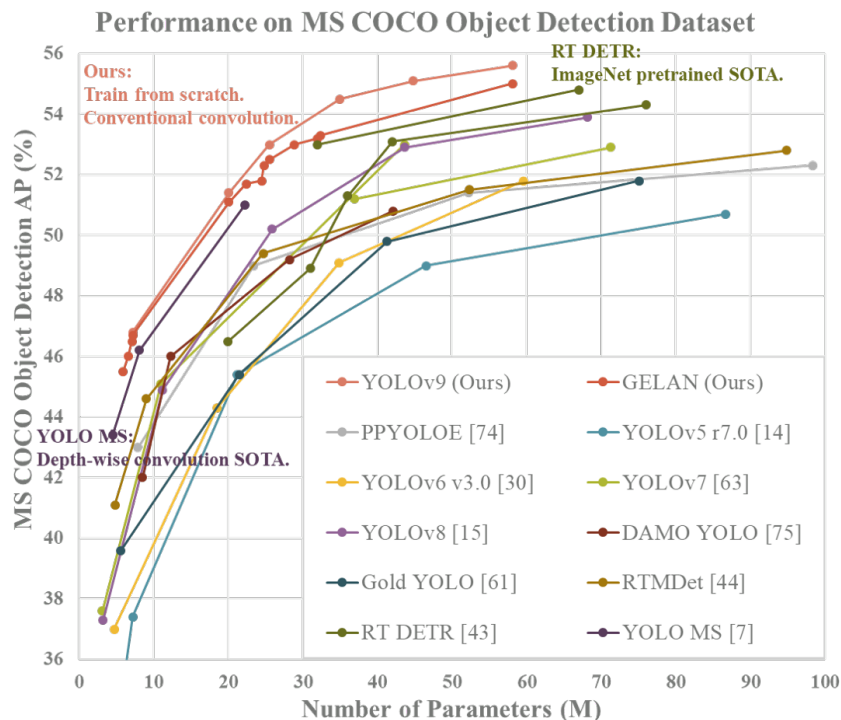


Figure 1.6: Performance of different YOLO versions on COCO Object detection dataset

Notably, YOLOv9 achieves an impressive AP score that passes 55 at its peak, surpassing all previous versions at around 58 parameters. This graph underscores the ongoing refinement and optimization of the YOLO algorithm, solidifying its reputation as a leading

choice for real-time object detection. By leveraging programmable gradient information and integrating novel architectural enhancements, YOLOv9 represents a significant leap forward in the field of computer vision, offering unparalleled performance and versatility in diverse operational scenarios.

1.3.5 Basics of YOLO algorithm

The YOLO algorithm distinguishes itself with its single-stage architecture, where classification and localization occur simultaneously, unlike two-stage architectures where these tasks are segregated. This characteristic not only ensures speed but also renders it highly suitable for real-time applications. To train the model effectively, it necessitates labelled images, where objects are delineated by bounding boxes, indicating both their presence and class. But how does YOLO process these bounding boxes?

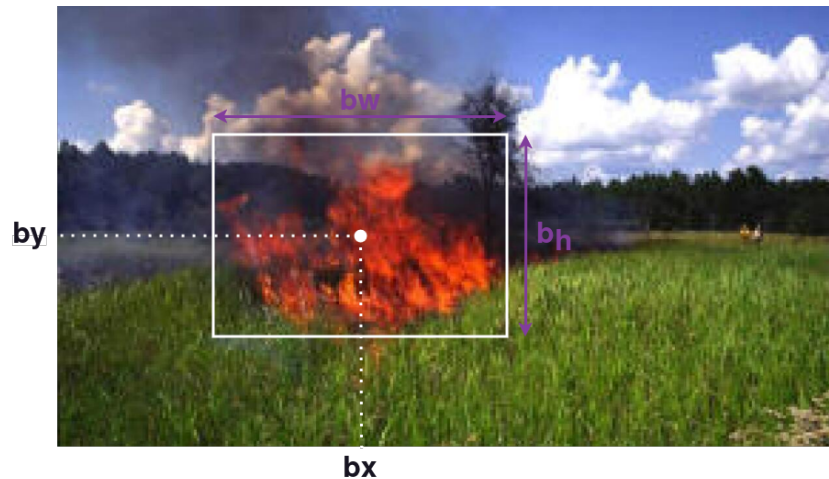


Figure 1.7: The parameters of object for localisation in yolo algorithm

Examining the illustrative images provided, considering it's a classification means that there is just one object in the image, we observe that the bounding box is defined by four parameters. Among these, b_x and b_y denote the coordinates of the box's centre, while b_w and b_h represent its width and height, respectively. Notably, all these parameters are normalised with respect to unity, minimising computational overhead during training. This normalisation conventionally sets the top left corner of the image as the origin $(0, 0)$ and the bottom right corner as $(1, 1)$. Importantly, the width and height can exceed unity, accommodating objects larger than the image itself. So, what does the output label resemble?

The output from our model is represented as a vector of size $(1, N)$, structured as follows: $Y = [P_c, b_x, b_y, b_h, b_w, C_1, C_2, \dots, C_n]$

Here, P_c denotes the confidence score indicating the presence of an object, while C_1 through C_n represent the confidences for each class.

To illustrate, let's consider a scenario where we're classifying images into two classes: fire and smoke. For instance, if we analyse an image and obtain the vector: $Y = [1, 0.4, 0.5, 0.5, 0.35, 0.95, 0.05]$, it implies that the image contains an object with a 95% probability of being classified as fire. Conversely, if no object is detected in the image, the vector assumes the form: $Y = [0, x, x, x, x, x, x]$. Here, $P_c = 0$ signifies the absence of an object, rendering the class confidences irrelevant. This vector-based representation provides concise and informative insights, facilitating efficient object detection and classification.

And now we introduce the loss function:

$$\text{loss} = \sum_{i=1}^N (Y_i - \hat{Y}_i)^2 \quad \text{if there is an object}$$

$$\text{loss} = (Y_1 - \hat{Y}_1)^2 \quad \text{if there is no object}$$

In object detection, YOLO introduces the concept of grid cells to detect multiple objects within an image. It partitions the image into grid cells and applies classification techniques to each cell. While the standard YOLO implementation divides images into a grid of 19×19 cells, for illustrative purposes, let's consider a simplified scenario with a 3×3 grid cell image.



Figure 1.8: illustration of grid image

Upon grid division and classification, we obtain a three-dimensional matrix. In the case of the aforementioned 3×3 grid, with two classes, the matrix dimensions would be $3 \times 3 \times 7$, where 7 corresponds to 1 (for P_c), 4 (for coordinates), and 2 (for classes). Consequently, we generate nine such vectors, each representing a grid cell.

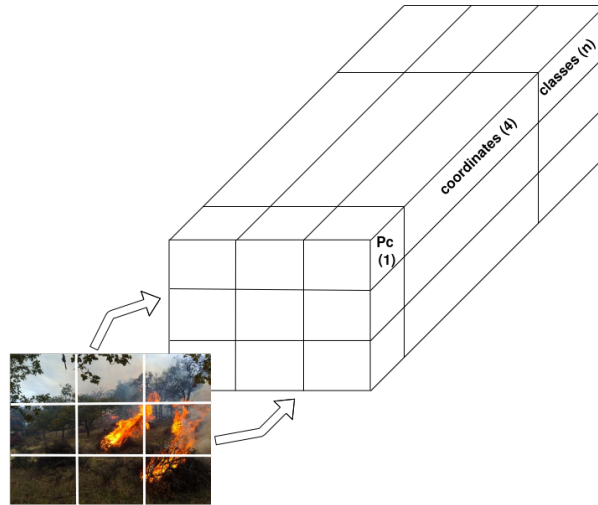


Figure 1.9: How result of the yolo model look like

In the actual YOLO algorithm, the matrix dimension extends to $19 \times 19 \times (5 + n)$. While this facilitates detection of one or fewer objects per grid cell, imagine a scenario with two objects in a single cell. Although YOLO provides mechanisms for such detection, it's worth noting its inherent limitation: the 19×19 grid cells theoretically allow for detection of up to 361 objects in a single image, yet we strive for more robust performance.

We can enhance object detection by incorporating anchor boxes tailored to specific classes, such as cars and people. Anchor boxes represent typical bounding box shapes associated with each class; for instance, cars often have a width larger than their height, while people may exhibit the opposite. By defining anchor boxes for each class, we enable the detection of multiple objects within a single grid cell simultaneously. However, a limitation arises when two objects share the same class or when two classes have identical anchor box shapes. Unfortunately, this approach doesn't align with our use case for wildfire detection, as neither fire nor smoke possess predefined shapes, particularly in natural environments. Therefore, while anchor boxes offer improvements in certain contexts, they are not applicable to our scenario due to the unpredictable shapes of wildfire-related objects.

1.4 Development Theory: Implementing a Dashboard with Flutter

The rapid advancement in drone technology has made it crucial to monitor their activities efficiently, especially in applications such as surveillance, agriculture, and delivery services. Concurrently, real-time weather monitoring is vital for ensuring safe and optimal drone

operation. This document details the development of a desktop GUI application designed to address these needs, utilizing Flutter for the user interface, GetX for state management and routing, and WeatherAPI for weather data.

1.4.1 Technologies Used

1.4.1.1 Flutter

Flutter is an open-source UI software development kit created by Google. It is used for developing applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia, and the web from a single codebase.

1.4.1.2 GetX

GetX is a powerful microframework for Flutter that provides features such as state management, dependency injection, and route management.

1.4.1.3 DIO

DIO is a powerful HTTP client for Dart, which supports Interceptors, FormData, Request Cancellation, File Downloading, Timeout, etc.

1.4.1.4 Clean Architecture

Clean Architecture is a software design philosophy that emphasizes separation of concerns, making the application easier to maintain, test, and scale.

1.4.1.5 svgImage

svgImage is a Flutter package that allows for the rendering of SVG (Scalable Vector Graphics) images, which are vector-based and can be scaled without losing quality.

1.4.1.6 StateMixin

StateMixin is a feature provided by GetX to manage different states of a widget, such as loading, empty, error, and data states, simplifying the process of state management before rendering the UI.

1.5 Theoretical Framework: Simulating Drone Operations with the Godot Game Engine

1.5.1 ForestWings

ForestWings is a simulator developed using the Godot Game Engine. Its primary purpose is to create a $4 \text{ km} \times 1 \text{ Km}$ virtual world featuring a road leading to a tunnel, surrounded by trees. We made ForestWings to test our patrolling algorithm, improve it, and finally tune it.

1.5.1.1 Dependencies

ForestWings simulator is developed mainly on GoDot Game Engine. Yet, it depends on some plugins which helped making the development process easier and faster.

- **Terrain3D**: helps making terrains, we used it to make realistic environment, so that we can tune our patrolling algorithm.
- **ScatterPlot**: facilitates dropping components randomly or according to a distribution algorithm in a portion of surface. We used this plugin to drop trees all around the four kilometers square surface of our world.

Even with these advanced plugins, our simulator won't be that realistic without using skins and Animated 3D models. Therefore, we downloaded 4K-Quality skins for dirt, mud, grass [1.11](#), rocks[1.13](#), trees[1.12](#), and drone[1.10](#) from polyhaven [\[10\]](#).



Figure 1.10: 3D Model of the used Tello drone.



Figure 1.11: Dirt skin used in our environment.



Figure 1.12: 3D Model of Tree.



Figure 1.13: Terrain texture skin.

1.6 State of the art

The integration of drone technology with artificial intelligence represents a significant advancement in emergency management, particularly in early fire detection and monitoring. Drones equipped with AI capabilities have transformed the traditional methods used by firefighting and surveillance teams, enabling them to detect the onset of fires and the presence of smoke with pinpoint accuracy and in the shortest possible time. This technological integration enhances operational efficiency by providing near-instant and real-time responses, significantly mitigating risks to human life and other resources. The use of AI-enabled drones is crucial in managing the complexities associated with wildfires and urban fires, ensuring the safety of emergency responders, and minimizing damage and potential loss of life through early and accurate fire detection [11] [12] [13].

Advances in drone technology have been pivotal in enhancing surveillance and monitoring capabilities, especially in critical applications such as fire detection. Modern drones are now equipped with sophisticated cameras and sensors, as well as robust communication systems that support real-time data sharing. High-resolution cameras provide clear and detailed images from aerial perspectives, enabling precise identification and assessment of fire-affected areas. Infrared sensors are essential for nighttime surveillance, allowing drones to detect heat signatures associated with fires even under low visibility conditions. Thermal imaging is critical for identifying hot spots and subtle temperature variations, enabling more accurate pinpointing of fire locations and real-time monitoring of flame spread, which is vital for effective firefighting resource allocation. These sensor technolo-

gies ensure that data captured by drones is comprehensive and actionable [11] [14].

The effectiveness of drones in emergency situations is further amplified by advanced communication systems that facilitate seamless autonomy and coordination. Real-time data transmission capabilities enable drones to send data instantly back to control centers for immediate analysis and decision-making, which is crucial during fire emergencies where every second counts. Autonomous operations allow modern drones to communicate with each other without human intervention, creating a mesh network in the sky. This network enables coordinated flight patterns, area coverage, and data sharing among multiple drones, increasing surveillance area and improving response effectiveness. These technological enhancements not only improve the capabilities of drones in detecting and monitoring fires but also ensure timely, data-driven interventions that are less reliant on ground-based operations [11] [12] [14].

Advances in artificial intelligence have significantly enhanced the functionality of drones, particularly in object detection and autonomous navigation [15]. The evolution of object detection algorithms such as YOLO, SSD (Single Shot MultiBox Detector), and Faster R-CNN (Region-based Convolutional Neural Networks) has improved drone's real-time detection capabilities. YOLO processes images in real-time, making it suitable for situations where speed is crucial, while Faster R-CNN offers higher accuracy through a region proposal network. SSD balances speed and accuracy, detecting multiple objects within an image using a single neural network. Additionally, advances in deep learning have propelled autonomous navigation, enabling drones to operate independently in complex environments. Algorithms based on deep learning and reinforcement learning allow drones to make decisions and navigate obstacles without human intervention. Recent developments in liquid neural networks, which adapt their structure dynamically, enhance drone's ability to perform in diverse and changing conditions, improving their effectiveness in tasks such as environmental monitoring, search and rescue.

Despite these advancements, challenges such as battery life and flight duration, as well as accuracy in diverse conditions, persist. Historically, limited battery life has constrained drone's operational time, limiting their usefulness in prolonged tasks like continuous fire monitoring. Additionally, drones must operate in various environmental conditions, including smoke, fog, and variable lighting, which can affect fire detection accuracy. Recent breakthroughs, however, are addressing these challenges. Innovations in battery technology, including developments in lithium-sulfur and solid-state batteries, promise higher energy densities and longer lifespans. Energy-efficient designs and power management

algorithms are being integrated to optimize power consumption, allowing for extended flight times and larger coverage areas. Improvements in AI algorithms have enhanced the robustness and accuracy of fire detection systems under varying conditions, with advanced machine learning models better distinguishing between smoke, fog, and other visual impediments, thereby enhancing detection reliability [11] [13] [14].

Future trends and research directions in AI-equipped drones for fire detection focus on machine learning optimization and collaborative systems. Advancements in machine learning aim to develop lightweight models that maintain high accuracy while reducing computational demands, crucial for deployment in drones with limited processing power and energy efficiency. Techniques such as model pruning, quantization, and knowledge distillation help streamline neural networks, enabling faster processing speeds and longer operational times. Collaborative systems, or swarm intelligence, involve multiple drones operating in a coordinated manner to cover larger areas more effectively. This approach enables comprehensive data collection and real-time monitoring across vast landscapes. Drones can communicate and share data instantaneously, creating a dynamic and adaptive network that optimizes surveillance and monitoring processes. Research is also exploring decentralized AI, where decision-making is distributed among drones rather than centrally controlled, increasing the robustness and scalability of operations in challenging environments like wildfire monitoring. These ongoing innovations promise more effective, efficient, and scalable solutions for fire detection and emergency response, potentially reducing response times and increasing accuracy in critical situations [16].

Chapter 2

Hardware and Software

2.1 Introduction

This chapter provides an overview of the hardware and software components essential for our project's realization. Our focus lies on the Tello drone as the primary hardware, while software tools like Torch, OpenCV, Flutter, and YOLO empower our AI and computer vision capabilities. Additionally, we utilize the Godot game engine for drone simulation. This selection of tools forms the foundation of our autonomous drone system, enabling robust functionality and validation of our algorithms. Through this exploration, we highlight the integral role of both hardware and software in achieving our project objectives.

2.2 Drone Tello

The Tello EDU [17] stands out as an exceptional programmable drone tailored for educational purposes. It provides a user-friendly platform for learning programming languages. Enhanced with the upgraded SDK 2.0 [18], the Tello EDU offers advanced commands and expanded data interfaces, empowering users with greater flexibility. Equipped with DJI's flight control technology and Electronic Image Stabilization support, the Tello EDU ensures stable flight performance and high-quality imagery. Whether it's orchestrating multiple drones to fly in a synchronized swarm or developing innovative AI functionalities, the Tello EDU makes programming an enjoyable and rewarding experience.

2.2.1 Technical characteristics

Being primarily designed for indoor use, the drone operates within a set of defined parameters that cater to its specific functionalities.

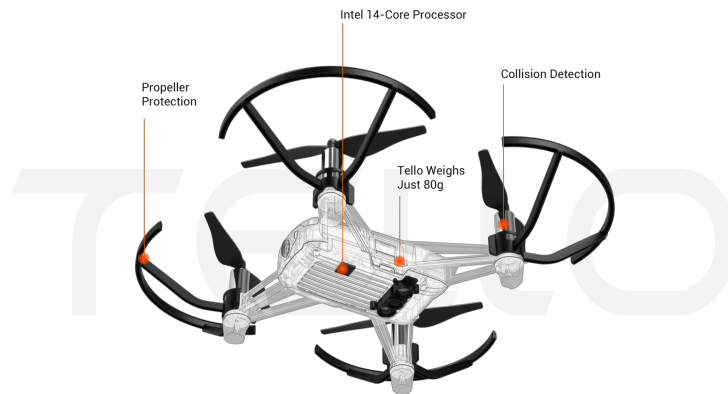


Figure 2.1: Drone Tello EDU

Its battery system offers remarkable convenience, facilitating easy replacement and charging. With the option to directly charge the battery within the drone using a USB cable or through a charging hub, power management becomes effortless.

Equipped with a high-quality camera capable of capturing HD video, the drone enables the development of various applications through image processing. This feature opens avenues for creative exploration and innovation.

The Vision Positioning System, comprising a camera and an infrared 3D module, provides precise localization capabilities within a range of 0.3 m to 30 m. Although optimal performance is achieved within the 0.3 m to 6 m range, ensuring reliable operation under varying conditions.

A Drone Status Indicator, integrated with an LED, offers real-time feedback on the drone's operational state, enhancing user awareness and control during flight operations.

The drone's field of view defines the observable area captured by its camera, facilitating comprehensive monitoring and surveillance tasks.

Furthermore, electronic stabilization techniques enhance image quality through electronic processing, ensuring steady and clear footage in various operating environments.

2.2.2 Tello SDK 2.0

The Tello EDU drone boasts a comprehensive software development kit (SDK), forming the cornerstone for the development of various applications discussed in this document. RYZE Technologies offers a user guide online, providing detailed instructions on essential aspects are elaborated upon in subsequent sections of this document, offering a deeper understanding and practical insights into the drone's functionality.

2.3 Torch

Torch [19] is an open-source machine-learning library and scientific computing framework comprising script language that binds the Lua programming language. It contains many powerful algorithms for deep learning and is appreciated by the research community for its efficiency, flexibility, and minimalistic, easy-to-understand syntax. Torch is highly acclaimed by the research community because of its powerful GPU support and the highly efficient handling of tensor operations that are at the core of most deep learning applications.

2.3.1 Torch’s Role in Reinforcement Learning

Reinforcement learning is a part of machine learning and an area in which agents learn to make decisions by taking action in the environment and receiving either a reward or a penalty for it. Torch is so helpful in reinforcement learning due to its strong capacity to deal with complex dynamic computational graphs that change with learning. The automatic differentiation in the framework helps implement various RL algorithms effectively. Building on top of Torch, the libraries further abstract and give tools that make the implementation of RL algorithms easier, making PyTorch the best choice for autonomous system implementors or researchers in, for example, drones.

2.3.2 Key Features of Torch Useful for The Project

Torch’s architecture allows for easy and fast prototyping of critical RL models, which is essential in developing and tuning algorithms for drone behavior. Key features include:

- **Tensor computation and dynamic neural networks:** Torch uses a dynamic computation graph known as the define-by-run scheme, which is particularly suited for projects where the learning algorithm itself changes as the drone learns from its environment.
- **GPU acceleration:** This enables the processing of large volumes of data and simulations at much higher speeds, significantly reducing the time required for training models.
- **Rich pre-built libraries:** Torch provides a comprehensive ecosystem of tools and libraries like TorchVision for computer vision, which can be leveraged for drone vision and navigation.

2.3.3 Advantages of Using Torch

Using Torch for drone autopilot algorithms offers several advantages:

- **Flexibility and Scalability:** The dynamic nature of Torch's computation graphs makes it incredibly flexible, allowing for adjustments and optimizations on-the-fly based on the drone's performance and learning progress.
- **Active Community and Rich Ecosystem:** Torch benefits from a vast community of developers and researchers, which results in a wealth of tutorials, open-source projects, and forums for troubleshooting. This community support is invaluable for rapidly evolving projects such as drone automation.

2.4 OpenCV

OpenCV [20] (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. It provides a common infrastructure for computer vision applications and accelerates the use of machine perception in commercial products. Being highly optimized and with an extensive library of programming functions, OpenCV supports a wide range of applications related to image processing, video capture, analysis, and face recognition.

2.4.1 Video and Image Processing Capabilities

OpenCV excels in real-time image processing, which is essential for the dynamic and visually complex tasks involved in drone monitoring. For our project, OpenCV is paired with CVzone to enhance visual outputs, enabling effective detection and monitoring of critical situations. Key functions include:

- **Image Filtering:** Enhances image quality captured by the drone's camera, crucial under varying lighting and environmental conditions.
- **Annotation and Visualization:** When the YOLO model detects fire or smoke, OpenCV and CVzone are used to draw bounding boxes around the detected areas. These visual annotations are crucial for reviewing and responding to the detections, as they allow operators to see and evaluate the detections in real time on a PC.

2.4.2 challenges and solutions

Implementing OpenCV in drone technology for fire detection presents several challenges and corresponding solutions:

- Challenges
 - Real-Time Processing: High-resolution video analysis demands significant processing power, which can strain drone systems.
 - Environmental Variability: Changes in weather and lighting can severely affect image detection reliability.
 - Algorithm Efficiency: There is a need for a careful balance between the computational speed and the accuracy of the image processing algorithms.
- Solutions
 - Hardware Optimization: Utilization of GPUs to enhance the processing capabilities of drones, allowing for faster image analysis.
 - Image Tuning: Dynamic adjustment of filters applied to video frames based on real-time environmental data helps maintain accuracy.
 - Use of Pre-trained Models: Implementing AI models trained on diverse datasets improves the system's robustness and adaptiveness to different fire scenarios.

2.4.3 Advantages of Using OpenCV

The choice of OpenCV for this application is supported by several advantages:

- Open Source and Community Support: Provides access to extensive resources and a community of developers, which facilitates troubleshooting and continuous improvement of the detection algorithms.
- Cross-Platform Compatibility: OpenCV can be integrated across various operating systems and platforms, making it highly versatile for drone systems.
- Performance: The library is optimized for real-time operations, essential for the immediate analysis required in drone surveillance and emergency response scenarios.

2.5 YOLO

YOLO (You Only Look Once) is a state-of-the-art real-time object detection system that revolutionizes the way visual data is interpreted by treating object detection as a single regression problem, directly from image pixels to bounding box coordinates and class probabilities. Unlike traditional methods that process parts of the image separately, YOLO looks at the whole image during training and testing, which allows it to predict objects with high speed and accuracy efficiently. This approach makes YOLO exceptionally suitable for scenarios requiring fast and reliable object detection, such as monitoring and identifying fire and smoke with drones.

For a detailed discussion of YOLO's underlying theory and its functionalities, please refer to the first chapter of this thesis.

2.6 Flutter

Flutter [21] is an open-source UI software development toolkit created by Google. It is used for developing applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia, and the web from a single codebase. Flutter uses the Dart programming language, also developed by Google, and provides a rich set of pre-designed widgets and tools that enable developers to create visually appealing and responsive user interfaces.

2.6.1 Key Features of Flutter

- **Single Codebase:** Write once and deploy to multiple platforms.
- **Rich Set of Widgets:** Provides a comprehensive library of customizable widgets that adhere to the design principles of both Material Design (Android) and Cupertino (iOS).
- **Hot Reload:** Allows developers to see the results of their changes almost instantly without restarting the application.
- **High Performance:** Compiles to native ARM code, uses GPU rendering, and performs efficient garbage collection.
- **Open Source:** Actively developed and maintained by Google, with contributions from the community.

2.6.2 Architecture

Flutter's architecture is composed of three main layers:

- **Framework:** Written in Dart, this layer provides the core building blocks like widgets, rendering, and animation.
- **Engine:** Written in C++, it provides low-level rendering using the Skia graphics library, as well as platform-specific code.
- **Embedder:** This is the platform-specific layer that allows Flutter to run on various operating systems. It handles the communication between the Flutter engine and the platform.

2.7 GoDot game engine

Godot is an open-source, cross-platform game engine released under the MIT license. Developed by the community and maintained by the Godot Engine organization, Godot provides a comprehensive set of tools to create both 2D and 3D games from a unified interface. Its flexibility and ease of use have made it a popular choice among indie developers and professionals alike.

2.7.1 Key Features of Godot

- **Scene System:** Godot uses a scene system that allows for a flexible and modular approach to game development. Scenes can be anything from a single object to an entire game level, and they can be nested and instanced to create complex hierarchies.
- **2D and 3D Support:** Godot offers robust tools for both 2D and 3D game development, with a rich set of features for each. The 2D engine is particularly well-regarded for its performance and ease of use.
- **Scripting:** Godot uses GDScript, a Python-like language designed for making game development intuitive. It also supports C#, VisualScript, and can bind to other languages via GDNative.
- **Cross-Platform Deployment:** Games made with Godot can be exported to multiple platforms, including Windows, macOS, Linux, Android, iOS, HTML5, and more.

- **Integrated Development Environment (IDE):** Godot comes with a fully integrated development environment that includes a scene editor, script editor, debugger, and a variety of other tools.
- **Animation System:** The engine has a powerful animation system that supports skeletal, sprite, and property animations. This makes it easier to bring characters and scenes to life.
- **Open Source:** Being open-source, Godot is free to use and modify, with a strong community contributing to its development and support.

2.7.2 Architecture

Godot's architecture is designed to be intuitive and flexible, featuring:

- **Node System:** The core of Godot's architecture is its node system. Nodes are the fundamental building blocks of a game in Godot. Each node has a specific function, and nodes can be arranged in a scene tree to create complex behaviors.
- **Resource System:** Resources in Godot, such as scripts, textures, and audio files, are handled in a consistent way, making asset management straightforward.
- **Signals:** Godot uses a signal system to allow nodes to communicate with each other in a decoupled way. Signals can be emitted and connected to methods to handle events and interactions.

Weight	87 g
Drone's Dimensions	98mm×92.5mm×41mm
Propeller	3 inches
Integrated Functions	Telemetric sensor Barometer LED Vision System Wi-Fi 2.4 GHz 802.11n Real-time streaming 720p
Port	USB battery charging port
Operating temperature range	from 0° to 40°
Operating frequency range	from 2.4 to 2.4835 GHz
Transmitter (EIRP)	20 dBm (FCC) 19 dBm (CE) 19 dBm (SRRC)
Maximum distance of flight	100 m
Maximum speed	8 m/s
Maximum flight time	13 min
Maximum flight height	30 m
Removable	Yes
Capacity	1100 mAh
Voltage	3.8 V
Type	LiPo
Energy	4.18 Wh
Net Weight	25 ± 2 g
Temperature range when charging	from 5° to 45°
Maximum Load Power	10 W
Photo	5 MP (2592x1936)
Field of view	82.6°
Video	HD 720p 30 fps
Format	JPG (Photo) MP4 (Video)
Electronic stabilization	Yes

Tableau 2.1: Technical characteristics of Tello

Chapter 3

Methodology/Project Work

3.1 Introduction

This chapter presents the comprehensive design approach for developing an autonomous drone system capable of predicting fire ignition probabilities, patrolling designated regions, and detecting wildfires using advanced computer vision technology. It details the architecture and components of the system, explaining their interactions to achieve the project's objectives.

The autonomous drone system is developed with several key functionalities in mind: gathering information from weather APIs to predict fire ignition probabilities as an initial indicator, utilizing robust computer vision for accurate wildfire detection, ensuring efficient autonomous navigation and control, enabling seamless data transmission, and providing an intuitive interface for real-time monitoring and visualization. Each component is meticulously integrated to ensure effective operation and compliance with the specified requirements.

The subsequent sections delve into the specifics of the system architecture, computer vision implementation, navigation and control algorithms, communication protocols, dashboard design, and security measures. Additionally, this chapter discusses the integration processes to form a cohesive and functional system. By providing detailed insights into the design and development process, this chapter aims to showcase the innovative approaches and technical rigor employed in building a state-of-the-art autonomous wildfire detection system.

3.2 System Architecture

This section provides a comprehensive overview of the project, detailing the flow of data and the functionalities implemented to detect and manage fire hazards using weather data, drone patrols, and computer vision. Refer to the accompanying diagram 3.1 for a visual representation of the system architecture.

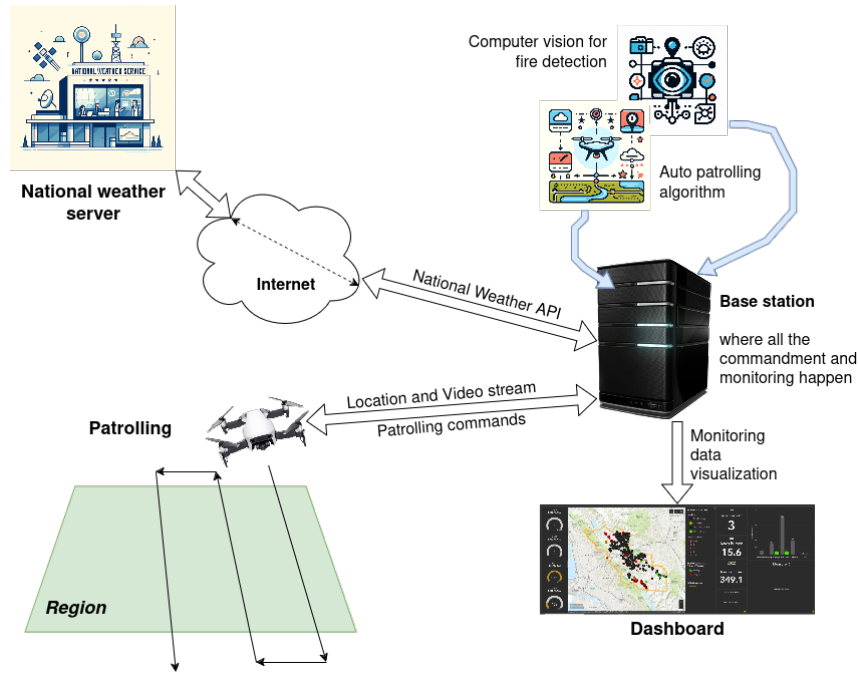


Figure 3.1: illustrative diagram for our project

The project begins with the collection of weather data from the National Weather API. Key parameters such as temperature, pressure, and humidity are retrieved. This data is crucial for assessing the environmental conditions that influence fire risk.

The collected weather data is visualized on a user-friendly dashboard, providing real-time insights into current conditions. Additionally, a probability calculation is performed using the weather data and initial knowledge of the region and season. For example, regions with deciduous trees that shed leaves in the spring have a higher probability of fire ignition. This calculation helps in assessing the likelihood of fire incidents.

Based on the calculated fire risk, the frequency of drone patrols over a region is adjusted. The drones have a predefined patrolling frequency, but this can be increased if the probability of fire is high. During patrols, drones send a continuous stream of location data and video footage.

The location data from the drones is fed into the patrolling algorithm, which determines the next path for the drone to follow. This ensures comprehensive coverage of the

region and optimizes patrol routes based on real-time data.

The video stream captured by the drones is processed by a computer vision model trained to detect fire. The live video is also displayed on the dashboard for monitoring purposes. If the model detects signs of fire, an alert is generated on the dashboard.

When a potential fire is detected, an alert is displayed on the dashboard. Simultaneously, notifications are sent to individuals and organizations involved in firefighting. To minimize false alarms and reduce costs associated with the fire detection model, a manual confirmation step is included. A human operator can review the alert and confirm the presence of a fire before any further action is taken.

The integrated system leverages weather data, drone surveillance, and computer vision to provide a robust solution for early fire detection and management. The diagram illustrates the flow of data and the interaction between various components, highlighting the systematic approach adopted in this project.

3.3 Computer Vision System

3.3.1 Computer Vision Algorithms Used for Wildfire Detection

YOLO (You Only Look Once) is a widely-used algorithm for object detection, with multiple versions released, ranging from YOLOv1 to YOLOv9. Although YOLOv9, released this year, boasts improved precision and mean Average Precision (mAP), it lacks an SDK, making its implementation in real-world projects challenging and potentially compromising performance or security. Therefore, we opted for the latest stable and well-documented version, YOLOv8, which provides all the necessary tools for project integration. Moreover, the performance difference between YOLOv8 and YOLOv9 is marginal.

In the development and tech realms, it is generally advised against using the very latest updates or versions in production environments, reinforcing our decision to use YOLOv8. In our project, we train our model on various datasets with different configurations, comparing them to determine the best performance. Our selection is not solely based on training results but also on extensive testing of the model using real fire videos and images sourced from the internet. This rigorous evaluation ensures the chosen model performs optimally in practical scenarios. For more information about the working principles of YOLO, please refer to Chapter .

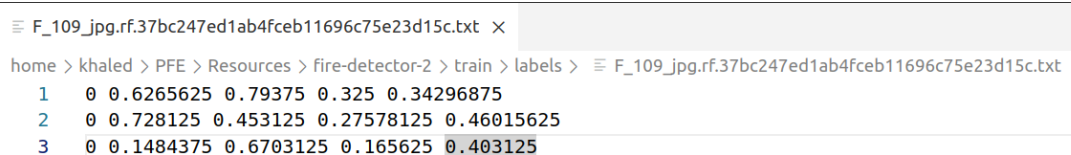
3.3.2 Data Preprocessing and Model Training

3.3.2.1 Data Preprocessing

As previously mentioned, our training dataset for fire and smoke detection was sourced from publicly available posts on the internet, specifically from well-recognized websites in the AI and machine learning community, such as Kaggle and Roboflow Universe. These platforms provide a wealth of high-quality data contributed by developers and annotators worldwide, who have made their datasets public.

We meticulously collected data representing various regions globally, ensuring our model can handle footage of different forests with diverse vegetation and coverage. This approach enhances the robustness and generalizability of our model, enabling it to perform effectively in a wide range of environmental conditions. By leveraging these diverse datasets, we ensure that our fire and smoke detection model is not only trained on a variety of scenarios but is also tested to perform reliably in real-world applications. This comprehensive dataset collection strategy is a crucial step in developing a model capable of addressing the complexities of fire and smoke detection across different geographical areas and vegetation types.

Every object detection algorithm employs its own method of annotation, typically using formats such as XML files, TXT files, or JSON files to label and categorize objects within images. The YOLO (You Only Look Once) algorithm is no exception to this. YOLO uses a specific annotation format that involves TXT files. Each image has a corresponding TXT file containing the annotations, where each line represents a single object in the image. The annotation format includes the class label, the coordinates of the bounding box's center (normalized to the image width and height), and the width and height of the bounding box (also normalized). This concise and efficient annotation method is integral to YOLO's ability to perform rapid and accurate object detection, ensuring that the data is structured in a way that optimizes the algorithm's performance during detection. Figure 3.2 is an example of this annotation.



```
home > khaled > PFE > Resources > fire-detector-2 > train > labels > F_109_jpg.rf.37bc247ed1ab4fceb11696c75e23d15c.txt
1 0 0.6265625 0.79375 0.325 0.34296875
2 0 0.728125 0.453125 0.27578125 0.46015625
3 0 0.1484375 0.6703125 0.165625 0.403125
```

Figure 3.2: Image labeled in YOLO format

For example, consider a TXT file snippet that contains three objects, all belonging to the same class (class 0). Each line in the file represents one object. The first number is the

class label (0), followed by two numbers indicating the center coordinates of the object within the image (normalized to the image's width and height). The final two numbers represent the object's width and height, also normalized. This annotation method ensures that the data is consistently structured, allowing the YOLO algorithm to efficiently and accurately detect objects in the image.

Since we are collecting data from various websites, the annotation formats differ significantly. To train our model effectively, we need to adjust and transform these datasets into a consistent YOLO format, which requires TXT files with specific annotation conventions. The data we collect may be in different formats, such as TXT or XML, or even in the YOLO format but with varying class labels. For example, in one dataset, class 0 might represent fire, while in another, class 0 might represent smoke. Given that our datasets contain thousands or tens of thousands of annotations, manual conversion is impractical. Therefore, we developed scripts to automate this process. Below, we provide an example of such a transformation, with additional examples and details provided in the Appendix.

In this example, we have annotations in the YOLO format from two different datasets, each representing fire and smoke but with different class labels. To merge these datasets into a cohesive training set, we developed a script 1. This script standardizes the class labels across both datasets, ensuring that the same class label is used for fire and smoke in each data set.

Algorithm 1 Python function to permit the classes label of a model.

```
def replace_starting_zero_with_three(folder_path):
    for filename in os.listdir(folder_path):
        if filename.endswith('.txt'):
            file_path = os.path.join(folder_path, filename)

            with open(file_path, 'r') as file:
                lines = file.readlines()

            modified_lines = ['3' + line[1:] if
                ↪ line.startswith('0') else line for line in
                ↪ lines]

            with open(file_path, 'w') as file:
                file.writelines(modified_lines)
```

The script iterates over all the files in the folder and changes the first character to 3 if it is 0. Subsequently, we do the same for the class label 1, changing it to 0. Finally, it

transforms the class label 3 to 1. This method effectively permutes the class labels, ensuring consistency across the datasets. This automated approach streamlines the process, allowing us to handle large datasets efficiently.

We can also run scripts to copy files from one folder to another, as shown below:

Algorithm 2 Python function to copy text files from a source folder to a destination.

```
def copy_txt_files(source_folder, destination_folder):
    if not os.path.isdir(source_folder):
        print(f"The source folder {source_folder} does not
        ↪ exist.")
        return

    if not os.path.isdir(destination_folder):
        os.makedirs(destination_folder)

    for filename in os.listdir(source_folder):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg',
        ↪ '.gif', '.bmp')):
            src_file_path = os.path.join(source_folder, filename)
            dest_file_path = os.path.join(destination_folder,
            ↪ filename)
            shutil.copy(src_file_path, dest_file_path)

    print("Text files copied successfully.")
```

This copying script is useful because of the folder structure that YOLO imposes in order to train our model, which will be explained in the following sections.

YOLO requires a YAML file that describes the data-set, including essential information such as the paths to training, testing, and validation folders. Each of these folders should be divided into two sub-folders: one for images and one for labeling TXT files, as illustrated in Figure 3.3. While the folder structure does not have to match the figure exactly, this layout is adopted for simplicity and clarity. Additionally, when splitting our data set, a common approach is to use 70% of our data for training, 20% for validation, and 10% for testing. This ensures a balanced distribution of data across the different stages of model development.

In our data set, we have gathered a total of 9,687 labeled images. We divided them as follows: 1,937 images for validation, 969 images for testing, and the remaining 6,781 images for training. This distribution ensures a balanced and effective allocation of data for each stage of model development.

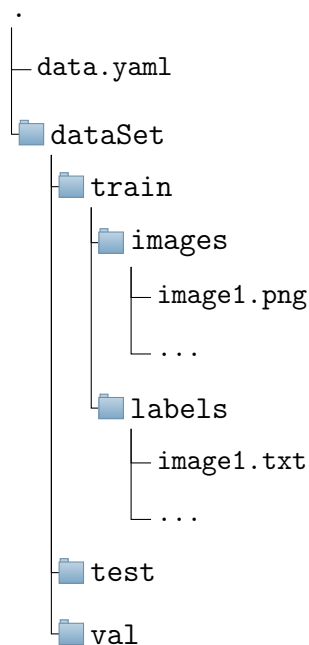


Figure 3.3: Folder structure of the dataset.

3.3.2.2 Fire and Smoke Detection Model Training

To train our model, we utilize the YOLOv8 repository developed by Ultralytics. Training the model demands significant computational power, making it impractical to train on a laptop without a GPU. We have two primary methods for training: using Google Colab or a powerful computer with a GPU. Initially, we used Google Colab, but encountered limitations with the free GPU runtime. Consequently, we transitioned to our school's server, which provided us with accounts and SSH access. This section illustrates both methods, highlighting their similarities and subtle differences.

Google Colab, or "Colaboratory," is a free, cloud-based platform provided by Google that allows users to write and execute Python code in a Jupyter notebook environment. It is particularly useful for tasks that require substantial computational resources. Colab provides free access to GPUs and TPUs, making it a valuable tool for data scientists and researchers who need to perform heavy computations without owning powerful hardware, like in our case.

The first step is to change the runtime to GPU runtime. We can verify if we are using the GPU by running the following command:

```
!nvidia-smi
```

Once we have confirmed that we are using the GPU runtime, we can proceed by

mounting Google Drive to Google Colab. This allows us to use Google Drive as storage for our dataset and training results, ensuring that our data is easily accessible and securely stored:

```
from google.colab import drive
drive.mount('/content/drive')
```

then we set our working directory as a base for our project using the os library:

```
import os
HOME = os.getcwd()
```

Next, we clone the Ultralytics repository or install it using pip:

```
!pip install ultralytics
!pip install -q roboflow
```

The Roboflow library simplifies the process of downloading and installing datasets, saving time and effort. Instead of downloading the dataset to your computer, performing preprocessing, and then uploading it to Colab, you can preprocess your data directly in Colab using the same methods discussed in the previous section.

Before proceeding, let's understand the parameters we adjust to train our model:

- **Epochs:** Represents a complete pass through the entire training dataset. During an epoch, the model processes each training example once through the following steps:
 - Forward Propagation
 - Backward Propagation
 - Weight Update

The concept of epochs is often used in conjunction with "batch size" and "iterations."

- **Batch Size and Iterations:** Batch size refers to the number of training examples used in one forward and backward pass. Using the entire dataset for each update may be computationally expensive, so it is divided into smaller batches. Iterations denote the number of batches needed to complete one epoch.

- **Close-mosaic:** Controls the behavior of Mosaic augmentation, particularly when and how this augmentation is applied during training. Introduced with YOLOv4, Mosaic Augmentation involves combining four different images into one, providing multiple contexts and enhancing model robustness. Each image occupies one quadrant of the resulting composite image.
- **imgsz:** Target image size for training. All images are resized to this dimension before being fed into the model, impacting model accuracy and computational complexity.

Considering the factors for choosing the number of epochs:

- **Dataset Size:** Larger datasets might require more epochs.
- **Learning Rate:** A lower learning rate might require more epochs.
- **Model Complexity:** More complex models might need more epochs to converge.

We chose 30 epochs to begin with, given our large dataset (around 10,000 images), and the relatively simple YOLO algorithm. We will observe the learning rate during training and adjust the number of epochs in subsequent training runs. We set the batch size to 96, so our model processes 100 batches of 100 images each to complete one epoch.

We keep the close-mosaic setting at its default value of 10, adjusting it later if the training results are not satisfactory.

We can train and fine-tune our model using one of YOLOv8's pre-trained weights, such as yolov8n, yolov8s, yolov8m, yolov8l, or yolov8x, ranging from nano to extra-large. For our training, we chose yolov8m and installed it from the official github repository using the following command:

```
!wget -P {HOME}/weights -q  
→ https://github.com/ultralytics/assets/releases/download/v8.2.0/yolov8m.pt
```

This command downloads the model and stores it in a new folder called weights.

We are now ready to launch the training. This can be done via the command line or a Python script. Using the command line:

```
!yolo task=detect mode=train model=yolov8m.pt  
→ data=/content/drive/MyDrive/archive/data.yaml epochs=30 imgsz=640
```

Or using a Python script:

```
import os
from ultralytics import YOLO

model = YOLO("yolov8m.yaml")

results = model.train(data=os.path.join(ROOT_DIR, "data.yaml"),
→ epochs=30, imgsz=640)
```

Since we are using Colab, it is convenient to save our data. One option is to save it in Google Drive.

Utilizing the free GPU provided by Google Colab can be challenging for long-running operations due to frequent interruptions during peak usage times. To address this issue, leveraging the school's server is a more reliable and efficient option. Accessing the server is straightforward using SSH:

```
ssh -p port userName@ipAddress
```

With the school's server, there is no need to use Google Drive for storage. All datasets and results are saved directly on the server's hard disk. We can easily transfer files between our laptop and the server using SSH commands.

To copy files from our laptop to the server:

```
scp -r -P port folderToCopy
→ userName@ipAddress:/home/userName/destinationFolder
```

To copy files from the server to our laptop:

```
scp -r -P port userName@ipAddress:/home/userName/sourceFolder .
```

Since our user account lacks administrative privileges, we cannot install software or execute commands that require `sudo`. However, Python is pre-installed on the server. To install Python packages, we create a virtual environment and use the associated `pip`:

```
python3 -m venv yolov8
```

Activate the virtual environment:

```
source yolov8/bin/activate
```

Now, we can install packages using `pip`:

```
yolov8/bin/pip install opencv-python
```

The above example demonstrates installing OpenCV using the `pip` of the Python virtual environment.

To utilize the NVIDIA GPU on the server for training, we need CUDA. Training with the GPU significantly reduces the time compared to using the CPU. Since installing CUDA requires administrative privileges, we requested the system administrator to install it.

These steps outline the process we followed to train our model on the school's server. The remaining procedures are identical to those used in Google Colab.

3.3.3 Performance Metrics and Accuracy of the Detection System

after launching our first train, we can observe the result and adjust in the parameter of training or the datasets accordingly to to improve the performance of our model.

yolo algorithm stores the result in a file called

```
HOME/runs/detect/train
```

Lets present an analysis of the fire detection model's performance, trained using the YOLOv8 algorithm. The analysis is based on various metrics including training and validation losses, precision, recall, mean Average Precision (mAP), and confusion matrices.

The training and validation losses are illustrated in Figure 3.4. The training loss metrics, including box loss, classification loss, and distribution-focused loss (DFL), exhibit a steady decrease over the training epochs, indicating progressive improvement in bounding box localization, object classification, and overall prediction precision.

The precision and recall metrics, along with the mAP at 50% IoU and the mAP averaged over IoU thresholds from 50% to 95%, are also shown in Figure 3.4. The precision

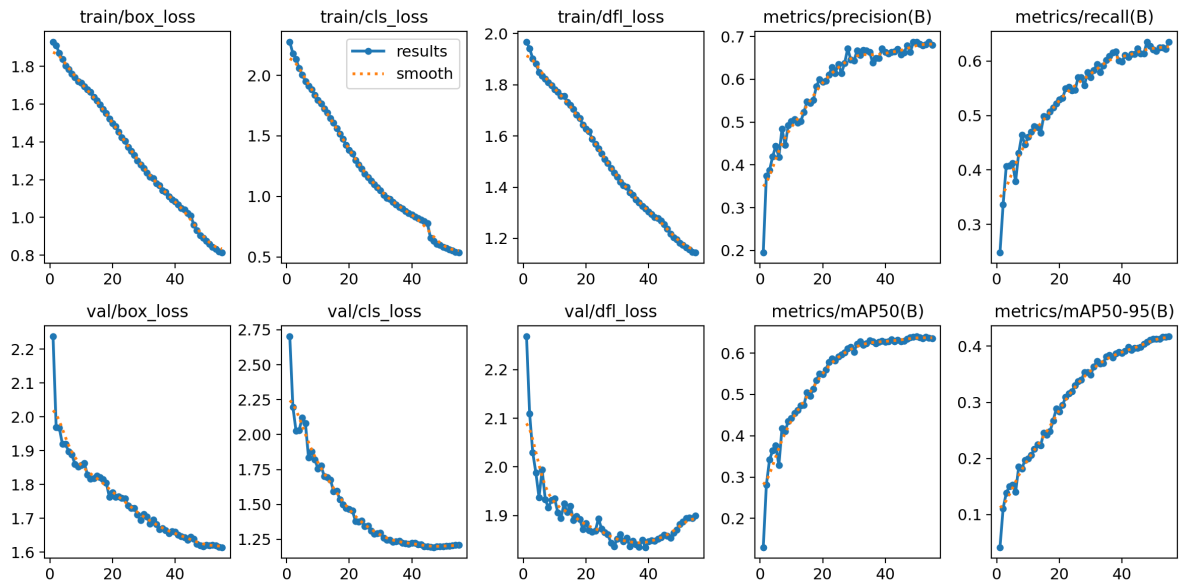


Figure 3.4: Training and Validation Losses for Box, Classification, and DFL, along with Precision, Recall, and mAP metrics.

starts from around 0.2 and reaches approximately 0.6, while recall increases from 0.3 to 0.7, indicating that the model is becoming more accurate and comprehensive in detecting true positive instances as training progresses.

Figures 3.5 and 3.6 depict the raw and normalized confusion matrices, respectively. The confusion matrix analysis reveals the following insights:

- **Fire:** The model correctly identifies fire instances with a high true positive rate (72%), but there are significant false negatives, indicating some fire instances are misclassified as background or smoke.
- **Smoke:** The true positive rate for smoke is moderate (61%), with some confusion between smoke and background, as well as a small number of smoke instances being misclassified as fire.
- **Background:** The model struggles most with background classification, showing a substantial number of false positives (classified as fire or smoke).

Figures 3.8 and 3.9 illustrate the precision-confidence and recall-confidence curves, respectively. The recall for fire is high at lower confidence thresholds, indicating the model's ability to detect most fire instances when the confidence threshold is low. However, the recall decreases as the confidence threshold increases. Precision increases with higher confidence thresholds for both fire and smoke, indicating more accurate predictions as the model becomes more confident.

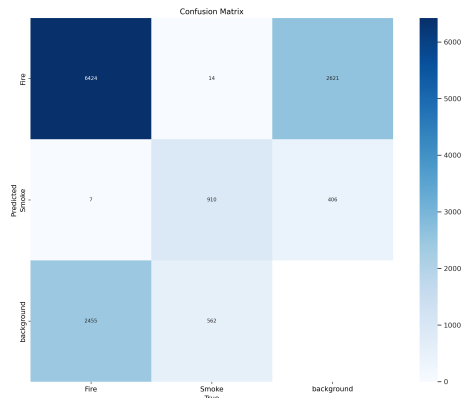


Figure 3.5: Confusion Matrix with Raw Values.

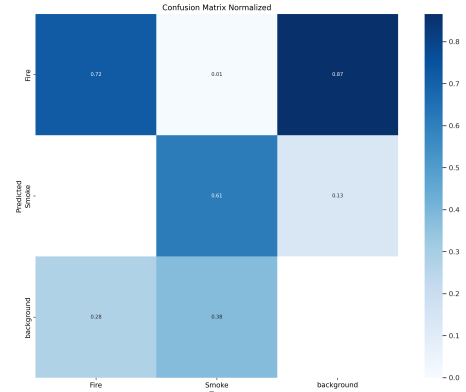


Figure 3.6: Normalized Confusion Matrix.

Figure 3.7: Confusion Matrix with Normalized and Raw Values.

The F1-confidence curve (Figure 3.10) demonstrates the balance between precision and recall. The F1 score peaks around a confidence threshold of 0.4-0.6 for both fire and smoke, suggesting this range as optimal for balancing precision and recall.

The YOLOv8 fire detection model shows promising performance with steadily improving metrics and decreasing losses. However, there are challenges in distinguishing smoke and background accurately. To further enhance model performance, it is recommended to increase the variety of smoke and background samples through data augmentation, experiment with different hyperparameters, explore different YOLOv8 architectures, and implement advanced post-processing techniques to refine predictions and reduce false positives.

The updated training and validation losses are illustrated in Figure 3.12. The training loss metrics, including box loss, classification loss, and distribution-focused loss (DFL), exhibit a continued steady decrease over the training epochs, indicating progressive improvement in bounding box localization, object classification, and overall prediction precision.

The precision and recall metrics, along with the mAP at 50% IoU and the mAP averaged over IoU thresholds from 50% to 95%, are also shown in Figure 3.12. The precision has improved, reaching approximately 0.8, while recall increases to about 0.85. This indicates that the model is becoming even more accurate and comprehensive in detecting true positive instances.

Figures 3.13 depict the normalized confusion matrices. The confusion matrix analysis reveals the following insights:

- **Fire:** The model now correctly identifies fire instances with a very high true positive

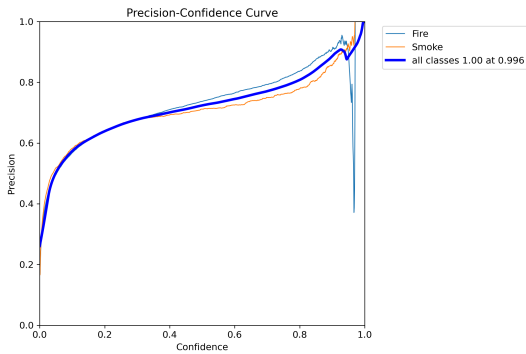


Figure 3.8: Precision-Confidence Curve for Fire, Smoke, and All Classes.

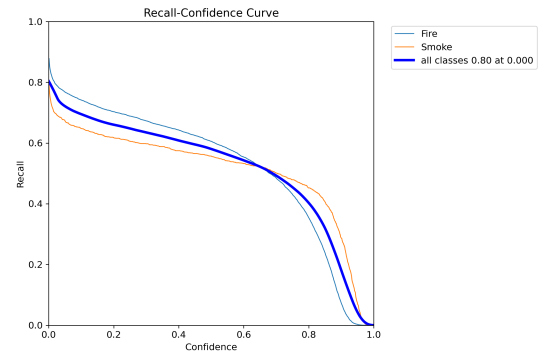


Figure 3.9: Recall-Confidence Curve for Fire, Smoke, and All Classes.

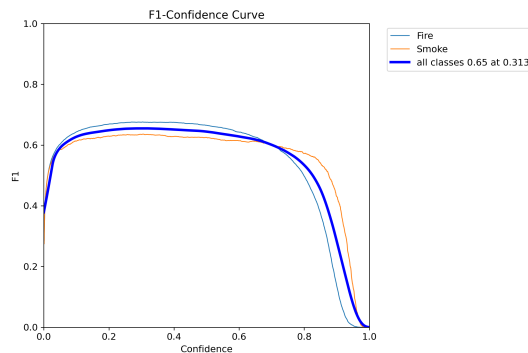


Figure 3.10: F1-Confidence Curve for Fire, Smoke, and All Classes.

Figure 3.11: precision and recall confidence curve

rate (97%), showing a significant improvement.

- **Smoke:** The true positive rate for smoke has increased to 96%, indicating better performance in distinguishing smoke from other classes.
- **Background:** The classification of background has also improved, with a reduction in false positives.

The improvements are further supported by the precision-confidence and recall-confidence curves (Figures 3.14 and 3.15), which show increased precision and recall at various confidence thresholds. The F1-confidence curve (Figure 3.16) demonstrates the balance between precision and recall, with the F1 score peaking at higher values compared to the previous model, suggesting an optimal confidence range for predictions.

Overall, the improvements made to the YOLOv8 fire detection model have resulted in enhanced performance metrics and reduced losses. While there is still room for further refinement, the model now performs its task effectively, demonstrating robust detection capabilities for fire and smoke instances.

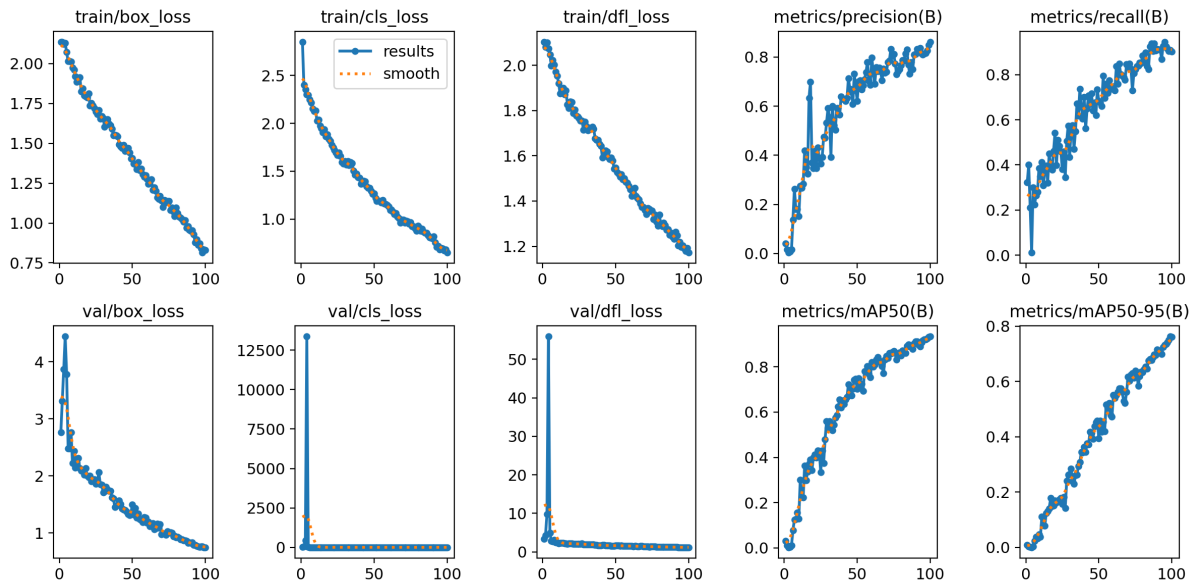


Figure 3.12: Updated Training and Validation Losses for Box, Classification, and DFL, along with Precision, Recall, and mAP metrics.

3.3.3.1 Pyromin detection Model Training

For person detection, we followed a similar training process as in fire and smoke detection but used a dataset tailored for person detection. Leveraging the pre-trained YOLOv8m model from Ultralytics, which performs exceptionally well on the COCO dataset, facilitated our fine-tuning efforts. The pre-trained model’s strong general object detection capabilities made it easier to achieve high accuracy in person detection.

Fine-tuning the YOLOv8m model on our dataset allowed us to achieve impressive results: 92% precision, 94% recall, 90% mAP, and 88% mAP50-95. These metrics demonstrate the model’s effectiveness in accurately identifying persons, meeting our project requirements.

Overall, using the pre-trained YOLOv8m model significantly simplified achieving excellent results in person detection, confirming the effectiveness of fine-tuning well-performing pre-trained models for specific tasks.

3.4 Autonomous Navigation and Control

Autonomous navigation and control are crucial components of the drone system, ensuring it can effectively patrol designated areas, avoid obstacles, and cover the entire region efficiently. This section details the path planning algorithms, real-time decision-making strategies for obstacle avoidance, and methods for ensuring efficient coverage of the patrol

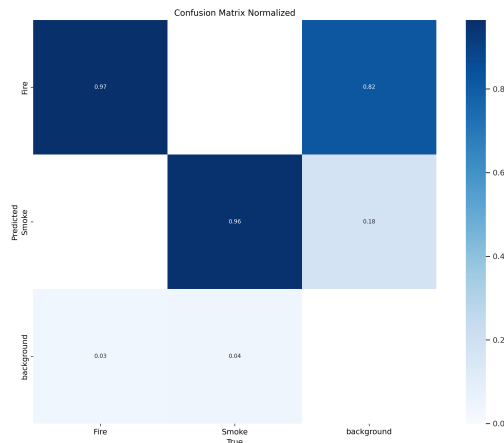


Figure 3.13: Updated Normalized Confusion Matrix.

area.

3.4.1 Path Planning Algorithms

for Path Planning Algorithm we used Deep Q-Networks, where we have trained our model for several epochs to find the optimum algorithm for path planning. Our model uses two main strategies:

3.4.1.1 Exploration-Exploitation

During its initial deployment, the drone employs a random patrolling strategy to explore the designated surface area thoroughly (Figure 3.18). Once the entire area has been covered, the drone maps the surface into a grid matrix with different values representing assumed rewards. These rewards are determined based on various criteria such as tree density, presence of pits, and terrain types (Figure 3.19).

Areas with higher reward values indicate regions that are more susceptible to fire risks compared to other spots. This exploration-exploitation approach allows the drone to prioritize and focus on high-risk areas for more frequent monitoring and patrolling, thereby enhancing the effectiveness of wildfire detection and prevention efforts.

This method can be used after the first deployment, and it can work very well under any circumstances. Yet, we use another method known as *Greedy method* which we will talk about next.

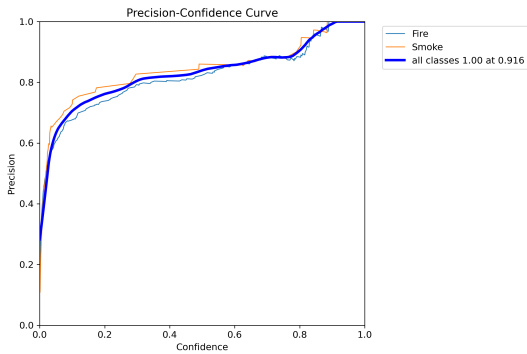


Figure 3.14: Updated Precision-Confidence Curve for Fire, Smoke, and All Classes.

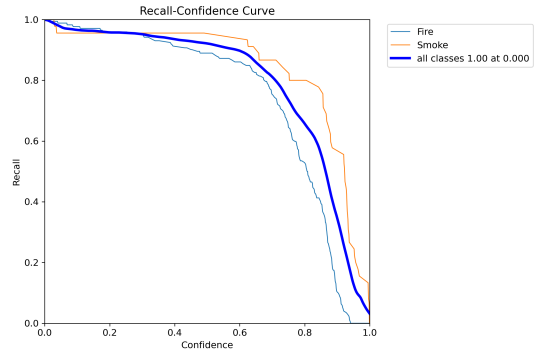


Figure 3.15: Updated Recall-Confidence Curve for Fire, Smoke, and All Classes.

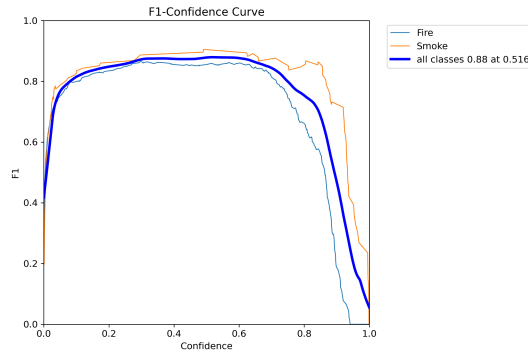


Figure 3.16: Updated F1-Confidence Curve for Fire, Smoke, and All Classes.

Figure 3.17: precision, recall and f1 confidence curve after update

3.4.1.2 Greedy Method

The greedy method is a more complex and nuanced approach where the drone attempts to maximize the number of rewards collected in a single pass. This strategy focuses on immediately targeting areas with the highest reward values, which represent regions with higher fire risk.

Benefits:

- **Focused Monitoring:** By prioritizing high-reward areas, the drone ensures that the most susceptible spots are observed frequently and thoroughly.
- **Efficient Resource Use:** This method makes efficient use of the drone's resources by concentrating efforts on the most critical areas, potentially leading to quicker detection of wildfires.

Drawbacks:

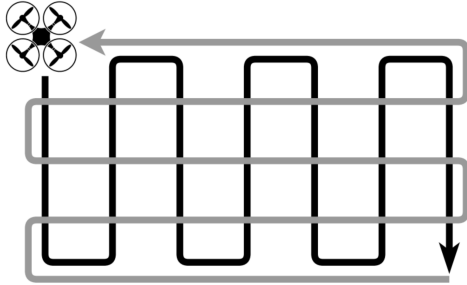


Figure 3.18: Exploration patrolling


2	1	6	1	1	1	0
1	1	1	3	1	1	1
1	1		1	1	1	1
6	6	1	4	1	1	1
6	6	1	1	9	1	1
1	1	2	1	1	1	1
0	2	2	2	1	1	0

Figure 3.19: Exploration Matrix

- **Potential Oversights:** The focus on high-reward areas might lead to neglect of lower-reward areas, which could still pose a risk.
- **Complex Path Planning:** The flight path under the greedy method is less straightforward and harder to predict compared to the systematic exploration approach. This complexity arises because the drone’s trajectory is continuously adjusted to maximize reward collection.

Despite its challenges, the greedy method is often prioritized because it aligns with the primary objective of ensuring that high-risk areas are monitored more frequently. By doing so, it enhances the likelihood of early wildfire detection and intervention, albeit at the cost of a more erratic and less predictable flight path.

To manage these complexities, advanced algorithms and real-time data analysis are employed to dynamically adjust the drone’s path, balancing the need for thorough coverage with the imperative of monitoring high-risk zones.

3.4.2 Real-time Decision-making for Obstacle Avoidance

Real-time decision-making is essential for the drone to navigate dynamically changing environments. The system must detect and respond to obstacles quickly to prevent collisions and ensure smooth navigation.

3.4.2.1 Sensor Fusion

We employ sensor fusion techniques, combining data from LIDAR, cameras, and ultrasonic sensors to create a comprehensive understanding of the surroundings. This multi-sensor approach allows the drone to accurately detect and map obstacles in its environment.

3.4.2.2 Dynamic Window Approach (DWA)

The Dynamic Window Approach (DWA) is used to calculate the optimal velocity commands for the drone by considering its kinematics and the need to avoid obstacles. The algorithm evaluates the possible velocities within a dynamic window, selecting the one that maximizes the objective function while ensuring collision avoidance.

Algorithm 3 Dynamic Window Approach (DWA)

```
1: Input: Current position  $p$ , current velocity  $v$ , obstacle map  $O$ , dynamic window  $W$ 
2: Output: Optimal velocity command  $v^*$ 
3: Initialize  $v^* \leftarrow (0, 0)$ 
4: Initialize maximum score  $S_{max} \leftarrow -\infty$ 
5: for each  $v_i \in W$  do
6:   Predict trajectory  $\tau$  from  $(p, v_i)$ 
7:   if  $\tau$  is collision-free in  $O$  then
8:     Compute objective score  $S(\tau)$ 
9:     if  $S(\tau) > S_{max}$  then
10:       $S_{max} \leftarrow S(\tau)$ 
11:       $v^* \leftarrow v_i$ 
12:     end if
13:   end if
14: end for
15: return  $v^*$ 
```

3.4.2.3 Reinforcement Learning (RL)

Another approach involves the use of machine learning, specifically reinforcement learning (RL). The RL model is trained in simulation to learn optimal navigation strategies through trial and error, receiving rewards for successful navigation and penalties for collisions. This model is then fine-tuned in real-world environments to adapt to specific scenarios and improve its decision-making capabilities.

Algorithm 4 Reinforcement Learning for Obstacle Avoidance

```
1: Initialize: Policy  $\pi$ , Q-function  $Q$ , environment  $E$ 
2: for each episode do
3:   Initialize state  $s \leftarrow E.\text{reset}()$ 
4:   for each step in episode do
5:     Select action  $a \leftarrow \pi(s)$ 
6:     Execute action  $a$  and observe reward  $r$  and next state  $s'$ 
7:     Update Q-function:  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
8:     Update policy  $\pi$  based on updated Q-function
9:      $s \leftarrow s'$ 
10:    if  $s$  is terminal then
11:      break
12:    end if
13:  end for
14: end for
```

By combining traditional algorithms like DWA with advanced machine learning techniques such as RL, the drone achieves a robust and flexible obstacle avoidance system capable of handling a wide range of scenarios in real-time.

3.4.3 Methods for Ensuring Efficient Coverage of the Patrol Area

Efficient coverage of the patrol area is achieved through systematic path planning and intelligent behavior algorithms. We use coverage path planning (CPP) algorithms, such as the Boustrophedon decomposition method, which divides the area into manageable subregions and generates paths that ensure complete coverage with minimal overlap.

Additionally, we integrate adaptive algorithms that adjust the drone's path based on real-time data, such as areas of higher fire ignition probability identified through weather API data. This dynamic adjustment ensures that the drone prioritizes critical regions while maintaining overall coverage efficiency.

To further enhance coverage, multiple drones can be deployed in a coordinated manner using swarm intelligence techniques. Each drone communicates with others to share information about covered areas and potential hazards, optimizing the overall patrol strategy and reducing redundant coverage.

3.5 Drone Communication and Video Stream

In this section, we detail communication with the drone, covering command sending, video stream reception, and applying fire, smoke, and person detection models on the captured frames.

3.5.1 Communication Protocol

We establish communication with the drone using a User Datagram Protocol (UDP) connection. The drone streams video data to a specified IP address and port, while control commands are sent back using a similar UDP connection.

Python's socket library facilitates sending commands to the drone. Here's a code snippet illustrating how to send a sequence of commands like takeoff, movement, and landing:

Algorithm 5 Python function to send a list of command to the drone.

```
# ... (Socket initialization and drone address)

commands = ["command", "takeoff", "land"]

def send_commands(commands):
    for command in commands:
        sock.sendto(command.encode(), drone_add)
        time.sleep(3)

send_commands(commands)
```

The code initializes a UDP socket, binds it to a local port, and transmits a predefined list of commands to the drone's IP address and port. Each command is encoded and sent with a delay between commands for proper execution.

The drone streams video data in UDP format to a specific IP address and port. We leverage OpenCV to capture this stream and process it frame by frame, as shown in the following code snippet:

Algorithm 6 Python function to capture video stream.

```
import cv2

stream = 'udp://@0.0.0.0:11111'
video = cv2.VideoCapture(stream)

while True:
    ret, frame = video.read()
    if ret:
        cv2.imshow('Drone Video Stream', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

video.release()
cv2.destroyAllWindows()
```

This code establishes a connection to the video stream, reads frames continuously, and displays them using OpenCV's 'imshow' function.

We employ the YOLOv8m model to detect fire, smoke, and people within the video stream. The following code snippet demonstrates applying these models to each frame:

Algorithm 7 Model integration with the stream.

```
import cv2
from ultralytics import YOLO

# ... (Stream and video capture initialization)

model = YOLO('./train8/weights/best.pt')
personModel = YOLO('yolov8n.pt')

while True:
    ret, frame = video.read()
    if ret:
        results = model(frame)
        results2 = personModel(frame)
        # Process detection results...
        cv2.imshow('Detection', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

video.release()
cv2.destroyAllWindows()
```

This code captures frames, applies the YOLOv8m model for fire and smoke detection, and a separate pre-trained YOLO model for person detection. Detected objects are visualized on the frames displayed in real-time.

By integrating these components, we establish a robust system for drone-based video streaming and real-time detection of fire, smoke, and people. The utilization of YOLO models ensures both high accuracy and speed, making the system suitable for practical applications in surveillance and emergency response scenarios.

3.6 Calculation of Fire Ignition Risk

Calculating the risk of fire ignition involves multiple factors, such as meteorological parameters (temperature, humidity), terrain, vegetation type, and vegetation density. Various fire risk indices have been developed based on these factors. These indices are often region-specific, and their suitability can vary depending on local conditions.

3.6.1 Fire Risk Indices

3.6.1.1 Meteorological Parameters and Vegetation Characteristics

Meteorological parameters like temperature and humidity, along with vegetation type and density, play a crucial role in assessing fire risk. Several indices have been developed to quantify fire risk based on these parameters.

3.6.1.2 Common Fire Risk Indices

Different studies have proposed various fire risk indices tailored to specific regions:

- **Keetch-Byram Drought Index (KBDI)**: Widely used in the USA, but found unsuitable for predicting forest fires in Georgia and Mississippi.
- **Modified Drought Index (DBDI)**: An adaptation of the KBDI for specific regions.
- **Fire Weather Index (FWI)**: Used in Canada for forest fire risk assessment.

Given our lack of expertise in meteorological science and the specific characteristics of Algerian forests, selecting the most appropriate index is challenging. Therefore, we opted for a simpler index to provide basic insights into fire ignition risk.

3.6.2 Simple Fire Danger Index

3.6.2.1 Parameters

The Simple Fire Danger Index relies on a minimal set of meteorological parameters:

- **Temperature (T)**: Measured in degrees Celsius ($^{\circ}\text{C}$).
- **Relative Humidity (H)**: Expressed as a percentage.
- **Wind Speed (U)**: Measured in kilometers per hour (km/h).
- **Threshold Wind Speed (U0)**: Ensures a non-zero fire danger rating even at zero wind speed.

3.6.2.2 Calculation of the Fuel Moisture Index (FMI)

The Fuel Moisture Index (FMI) is calculated using the formula:

$$\text{FMI} = 10 - 0.25(T - H)$$

3.6.2.3 Calculation of the Simple Fire Danger Index (F)

To calculate the Simple Fire Danger Index (F), the following formula is used:

$$F = \frac{\max(U_0, U)}{\text{FMI}}$$

where:

- U_0 is the threshold wind speed.
- U is the wind speed.

3.6.3 Implementation

3.6.3.1 Python Code

The implementation of the Simple Fire Danger Index grammatically is straightforward following these steps:

- Calculate the FMI.
- Determine the maximum of the wind speed and the threshold wind speed.
- Calculate the Simple Fire Danger Index.

3.6.4 Fire Danger Levels

The Simple Fire Danger Index provides a scale to assess the level of fire risk:

Simple Fire Danger Index (F)	Fire Risk
[0, 0.7]	Low
[0.7, 1.5]	Moderate
[1.5, 2.7]	High
[2.7, 6.1]	Very High
$F > 7$	Extreme

Tableau 3.1: Simple Fire Danger Index potential scale.

The Simple Fire Danger Index, while basic, provides a useful tool for assessing fire ignition risk based on key meteorological parameters. This approach, using straightforward calculations and Python implementation, offers an accessible method for evaluating fire danger levels in regions where more complex indices may be impractical to apply. For the prototype, this index was implemented to provide initial insights. However, it will be adapted to better suit Algerian weather and terrain conditions through consultations with meteorological science specialists.

3.7 Dashboard Interface and Visualization

The Dashboard interface serves the purpose of monitoring and commanding the drone remotely. It comprises six main components:

- **App Bar:** Located at the top of the screen and spanning the entire width, the App Bar displays critical information including the *location*, *drone state*, and *risk* level.
- **Drone State Sidebar:** Positioned on the left side of the screen, this sidebar provides real-time updates on the drone's *Battery percentage*, *Altitude*, *Speed*, and *Barometer* readings.
- **The Map:** This section, reminiscent of a Google Maps interface, displays the virtual environment's map. It includes control buttons for *Zoom in*, *Zoom out*, and *current location* to facilitate navigation and situational awareness.
- **Tello Control:** This area allows for remote operation of the drone with shortcuts for essential commands such as *take off*, *land*, *stream*, and *path planning*.
- **Weather:** A dedicated column of widgets that presents comprehensive weather data, including *temperature*, *pressure*, *precipitation*, and *wind speed*, ensuring that users are informed of the environmental conditions that may affect drone operations.
- **Graphs:** This section features two key graphs: *Risk Prediction* and *Alert Monitoring*, complete with legends for clear interpretation. These graphs provide insights into potential risks and ongoing alerts, enhancing situational awareness and decision-making.

3.8 Project Integration

3.8.1 Integrating the Weather API into the Dashboard

In order to receive real-time weather and forecast data on our platform, we integrated the WeatherAPI [22]. This API provides free weather and forecast data and offers a Pro Plus free trial for 14 days. To effectively use this API, we first studied its responses to understand the data models it provides.

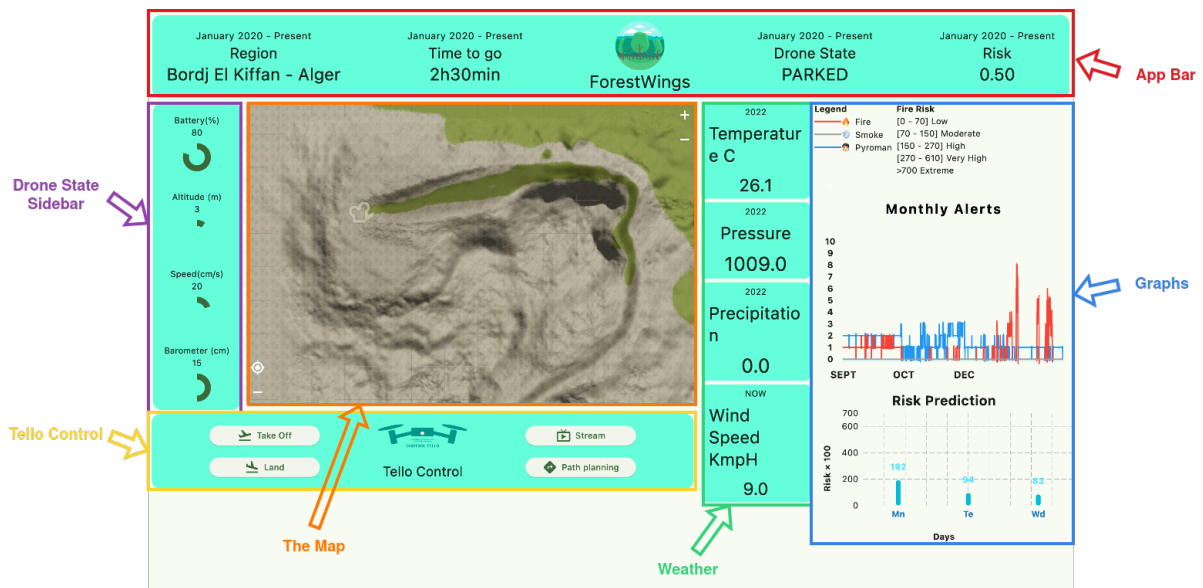


Figure 3.20: Dashboard Interface

3.8.1.1 Modeling data

The primary response from the WeatherAPI includes weather and forecast data encapsulated in the following Dart classes:

Algorithm 8 Weather Response Class Definition

```
class WeatherResponse {
  final Location location;
  final Current current;

  WeatherResponse({
    required this.location,
    required this.current,
  });
}
```

As shown, the `WeatherResponse` class includes `Location location` and `Current current`.

- The `location` field contains essential information about the geographic region, detailed in the `Location` class is given in the appendix 20
- The `current` field contains real-time weather data crucial for our application, structured as class provided in the appendix 21:

In these classes, we utilized the `json_annotation` package to facilitate the serialization and deserialization of the API responses. This approach ensures that our Dart models

accurately reflect the structure of the data provided by the WeatherAPI, making it easier to integrate and use within our platform.

3.8.1.2 Requesting and Handling Data

After successfully modeling the data, the next step is to start sending requests to the API and processing the responses. But how do we achieve this?

Flutter provides a powerful HTTP networking package named `dio`. This package supports a range of features including global configuration, interceptors, form data, request cancellation, file uploading/downloading, timeouts, custom adapters, and transformers, making it highly versatile for network operations.

To use this package according to best practices and adhere to the clean architecture principle, we have created a `WeatherService` class. Below is the implementation: As

Algorithm 9 Weather Service Class Definition

```
class WeatherService {
  WeatherService._();
  static final Dio _dio = Dio();
  static const String _weatherUrl =
    "http://api.weatherapi.com/v1/current.json?key=API_KEY&q=36.7725_
↪ 866,3.232888&aqi=no";
  static const String _forecastUrl =
    "http://api.weatherapi.com/v1/forecast.json?key=API_KEY&q=36.772_
↪ 5866,3.232888&days=7&aqi=yes&alerts=no";

  static Future<WeatherResponse?> getWeather() async {
    Response weatherData = await _dio.get(_weatherUrl);
    return WeatherResponse.fromJson(weatherData.data);
  }

  static Future<ForecastResponse?> getForecast() async {
    Response forecastData = await _dio.get(_forecastUrl);
    return ForecastResponse.fromJson(forecastData.data);
  }
}
```

shown, there are two static methods in the `WeatherService` class: `getWeather()` and `getForecast()`. These methods can be called without creating an instance of the class. To ensure that no instances of the class are created (as they are unnecessary), we have made the constructor private by adding `WeatherService._()`;

Next, we need the application to periodically call the API to retrieve real-time data. For this, we have created a **Stream** in the **Controller** of the home screen of the app. This **Stream** calls the `getWeather()` API endpoint every minute:

```
Stream<WeatherResponse?> weatherStream = Stream.periodic(  
    const Duration(minutes: 1), (_) async => await  
    ↪ WeatherService.getWeather(),  
).asyncMap((event) async => await event);
```

Similarly, we have set up a **Stream** for the forecast endpoint:

```
Stream<ForecastResponse?> forecastStream = Stream.periodic(  
    const Duration(minutes: 1), (_) async => await  
    ↪ WeatherService.getForecast(),  
).asyncMap((event) async => await event);
```

By leveraging these **Streams**, we ensure that the application consistently fetches and updates with the latest weather and forecast data in real-time, providing a seamless and responsive user experience.

3.8.1.3 Automating the Tasks

Up to this point, we have covered almost everything needed to set up our app with the WeatherAPI. However, there is one crucial aspect still missing: *Automation*. We want our app to automatically fetch data every minute and display it to the user seamlessly.

GetX provides an elegant solution for this requirement through its **Controller** lifecycle management. When a **Controller** is loaded, the `onReady()` function is invoked. This is where we can place our business logic to ensure that data fetching begins as soon as the screen is loaded.

Here is how we can achieve this:

Firstly, we need to bind the **Stream** to observable variables. We declare two observable variables as follows:

```
late Rx<WeatherResponse?> weather;  
Rx<ForecastResponse?> forecast = Rx<ForecastResponse?>(null);
```

The first variable, `weather`, is for handling the weather response. It is declared as a `late` variable, indicating that it will be initialized later. The second variable, `forecast`, is for handling the forecast response and is initialized with `null`.

The rationale behind this initialization strategy is that the screen will not render if we don't receive a weather response. On the other hand, the forecast data is not as critical and can be received later.

Next, we need to bind the **Streams** to these observable variables within the `onReady()` method of the Controller:

Algorithm 10 `onReady` Function Definition

```
void onReady() async {
    // Bind streams to observables
    weather.bindStream(weatherStream);
    forecast.bindStream(forecastStream);
    // Any additional setup code
}
```

By binding the streams in this manner, we ensure that our app starts receiving data automatically every minute as soon as the screen is loaded. The `onReady()` method is the perfect place to start this automatic data fetching because it is called once the Controller is fully initialized and ready.

Here's a more complete example of how the Controller might look:

Algorithm 11 `WeatherController` Class Definition

```
class WeatherController extends GetxController {
    late Rx<WeatherResponse?> weather;
    Rx<ForecastResponse?> forecast = Rx<ForecastResponse?>(null);

    @override
    void onReady() async {
        super.onReady();
        // Start data fetching
        weather.bindStream(weatherStream);
        forecast.bindStream(forecastStream);
    }

    // Additional methods and logic
}
```

With this setup, our app is now equipped to automatically fetch and display weather data every minute, providing users with real-time updates without any manual intervention.

By leveraging GetX's Controller lifecycle and reactive programming features, we can ensure that our app remains efficient and responsive, enhancing the overall user experience.

3.8.2 Integrating the Model and Alarm Launching

We've successfully trained our fire and human detection model, leveraging OpenCV, YoloV8, and the ultralytics framework. The model effectively logs every instance of fire or human presence it identifies. Now, the crucial step is integrating this powerful tool into our platform.

To bridge this gap, we've chosen Flask, a lightweight yet versatile Python framework, to create an HTTP server. Flask's streamlined nature aligns perfectly with our objective. This server will act as a central hub, continuously logging fire and human detection events.

On the platform side, we'll leverage Flutter's robust DIO package to make periodic HTTP requests to the server. This ensures we receive real-time updates on fire and human detections. Finally, Flutter's built-in push notification API allows us to display these critical alerts directly on the user interface, enabling swift response and enhanced safety.

Let's delve deeper into the specifics of this integration process:

3.8.2.1 Setting Up an HTTP Server Using Flask

Creating an HTTP server using Flask is one of the simplest tasks in backend development due to Flask's minimalistic and easy-to-use nature. Below is an example of setting up a basic Flask application.

Algorithm 12 Flask Application Example

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/detect")
def sendModelInfo():
    # We receive data from the detect() function
    data = detect()
    # Sends data as JSON
    return jsonify(data)

if __name__ == "__main__":
    app.run(debug=True)
```

As demonstrated in Algorithm 12, the server receives the necessary data from our model through the `detect()` function. This data is then formatted as JSON, which is the most common and preferred format for web applications due to its simplicity and wide compatibility.

To summarize, the steps involved are:

1. Import the necessary Flask modules.
2. Create a Flask application instance.
3. Define a route (`/detect`) to handle incoming requests.
4. Implement the logic to receive data from the `detect()` function and return it as a JSON response.
5. Run the Flask application in debug mode for development purposes.

With these steps completed, your HTTP server is ready to handle requests and respond with data in JSON format.

3.8.2.2 Setting Up Fire Alarm Services

Following the principles of clean architecture, we create a class with static methods to handle responses from the fire alarms server. This design ensures modularity and ease of maintenance.

Algorithm 13 Fire Alarms Class

```
class FireAlarmService {
    FireAlarmService._();
    static final Dio _dio = Dio();

    static const String _apiUrl = "${API_ALARMS_URL}/detect";

    static Future<FireAlarmModel?> getAlarm() async {
        Response alarmResponse = await _dio.get(_apiUrl);
        return FireAlarmModel.fromJson(alarmResponse.data as
↪ Map<String, dynamic>);
    }
}
```

As shown in Algorithm 13, the `FireAlarmService` class utilizes the Dio library to make HTTP requests. Key points include:

1. The class is defined with a private constructor to prevent instantiation, promoting the use of static methods.
2. A static instance of Dio is created to handle HTTP requests.
3. The API URL for fire alarm detection is stored as a static constant.
4. The `getAlarm` method asynchronously fetches data from the fire alarm server and parses the JSON response into a `FireAlarmModel` object.

This approach ensures that the fire alarm services are well-encapsulated and can be easily maintained and extended in the future.

“latex

3.8.2.3 Displaying Push Notifications

To display push notifications, we have created customized `SnackBars`, categorized into four distinct types:

```
enum CustomSnackBarType { INFO, SUCCESS, FAIL, ALERT }
```

Each `SnackBar` follows a uniform structure, differing only in color and icon based on its type:

Algorithm 14 `SnackBar` Global Schema

```
_defaultSnackBar(String title, String message, Color? color, IconData?  
↪ icon) =>  
Get.snackbar(  
title,  
message,  
backgroundColor: color,  
icon: Icon(  
icon,  
size: 30.sp,  
),  
maxWidth: 400.w,  
padding: EdgeInsets.symmetric(horizontal: 15.w, vertical: 10.h),  
duration: const Duration(seconds: 5),  
margin: EdgeInsets.only(top: 10.h),  
);
```

The `_defaultSnackBar` function is a global schema for our notifications. This function takes in a title, message, optional color, and icon to display a customized `SnackBar` using the `Get.snackbar` method. The customization includes setting the background color, icon, size, width, padding, duration, and margin.

To integrate these notifications, we utilize streams linked to observable variables, similar to the process in the weather API integration described in Section ???. However, in this instance, we employ the `ever` hook from the `GetX` package.

The `ever(observable, callback)` hook takes two parameters: the observable variable and the callback function. Whenever the observable variable's value changes, the callback function is triggered, allowing us to display the appropriate `SnackBar`.

Here is how we set up the `ever` hook:

Algorithm 15 Setting Up `ever()`

```
@override
void onReady() async {
    // code goes here
    ever(latestAlarm, showAlert);
    super.onReady();
}
```

In the above setup, the `ever` hook is used within the `onReady` method. When the observable variable `latestAlarm` changes, the `showAlert` function is called, displaying the appropriate push notification.

In summary, our approach to displaying push notifications involves defining custom `SnackBar` types, creating a global schema for `SnackBars`, and using the `ever` hook from `GetX` to reactively display notifications based on changes in observable variables.

3.8.3 Integrating the auto patrolling algorithm into the simulator

After training the model, the next step is to integrate it into our simulation environment. The trained model is designed to output a sequence of movement commands, such as `move_forward`, `move_left`, and other similar directives that define the patrol path.

To effectively utilize these movement commands in our simulation, we proceed with the following steps:

1. **Generate Movement Commands:** The trained model outputs a sequence of movement commands. Each command represents an action that the patrolling entity

should perform. These actions are typically based on the environment’s state and the desired patrolling behavior.

2. **Create a Path3D in Godot:** In Godot, a `Path3D` node is used to define a three-dimensional path. We use the series of movement commands generated by the model to create this path. The `Path3D` node allows us to specify a series of points in 3D space that the patrolling entity will follow.
3. **Convert Movements to Path Points:** Each movement command from the model corresponds to a specific change in position or direction. We convert these commands into a series of points (or “dots”) in 3D space, effectively plotting the patrol route. These points are added to the `Path3D` node to form a continuous path that the patrolling entity will navigate.
4. **Use PathFollow3D for Movement:** To make the patrolling entity follow the defined path, we use a `PathFollow3D` node. The `PathFollow3D` node is designed to move a `CharacterBody3D` (or any other object) along a `Path3D`. By attaching the `PathFollow3D` node to our `Path3D`, we ensure that the patrolling entity follows the generated path precisely.
5. **Attach the CharacterBody3D:** Finally, we attach the `CharacterBody3D` (which represents the patrolling entity) to the `PathFollow3D` node. This setup ensures that as the `PathFollow3D` node moves along the path, the `CharacterBody3D` will follow, thereby executing the patrolling behavior as determined by the model.

By following these steps, we effectively integrate the auto-patrolling algorithm into the simulator, enabling the patrolling entity to move according to the model’s outputs. This integration allows for dynamic and realistic patrolling behaviors that can adapt to different environments and scenarios within the simulation.

Overall, the combination of the trained model, `Path3D`, and `PathFollow3D` in Godot provides a powerful framework for implementing complex patrolling behaviors in a 3D simulation environment.

3.8.4 Integrating the map of the region in the dashboard

For this part, we exported a 2D map of the world from the ForestWings Simulator as a large PNG file. To enable interactive functionality, we used the `InteractiveViewer` widget, which allows users to zoom in and out of the map seamlessly.

Additionally, we added three buttons: **Zoom In**, **Zoom Out**, and **Current Position**. These buttons provide intuitive controls for navigating the map.

- **Zoom In:** This button zooms in on the map with a factor of 1.2.
- **Zoom Out:** This button zooms out of the map with a factor of 0.8. To ensure that the image does not shrink smaller than the wrapping box, we added a condition.
- **Current Position:** This button takes the user back to the position of the drone. The function responsible for this behavior is a generic function that can take you to any specified position.

```
zoomIn() {  
    scaleFactor = mapController.value.getMaxScaleOnAxis() * 1.2;  
    mapController.value = Matrix4.identity()..scale(scaleFactor);  
}
```

```
zoomOut() {  
    if (scaleFactor > 1.2) {  
        scaleFactor = mapController.value.getMaxScaleOnAxis() *  
↪ 0.8;  
        mapController.value =  
↪ Matrix4.identity()..scale(scaleFactor);  
    }  
}
```

```
focusOn(double x, double y) {  
    mapController.value = Matrix4.identity()..translate(x, y);  
    mapController.value = Matrix4.identity()..scale(scaleFactor);  
}
```

3.9 Future Enhancements and Scalability

3.9.1 Potential Improvements to the System Design

Our system design has primarily been influenced by the material available, the reliance on WiFi technology, and the limited computational resources integrated into the Tello drone. These constraints imposed significant limitations on our use cases. Consequently, we opted to transmit all data gathered by the drone to a base station—specifically, our laptop—for processing.

Moreover, the Tello drone lacks sensors for obstacle detection. While we have the capability to use its camera to train a model for obstacle detection, this approach would further burden the base station with additional computational load. This is not ideal given the existing constraints.

A more robust project design could be proposed if we had access to a professional-grade drone equipped with substantial computational resources. Such a drone would be capable of running reinforcement learning algorithms directly onboard. This would enhance the rapidity of action and alleviate the computational load on the base station. By offloading the processing tasks to the drone, we could achieve a more efficient and responsive system. This approach would eliminate concerns about connectivity interruptions, preventing the drone from getting lost or needing to return to the base station in the middle of a mission. This ensures continuous and reliable operation throughout the mission.

3.9.2 Plans for Scaling the System to Cover Larger Areas or Different Types of Environments

Patrolling large areas with a single drone presents significant challenges. Even an advanced drone may struggle to operate smoothly over expansive zones due to limitations in range and battery life. Additionally, communication issues arise from varying terrains, especially in our project, where the drone is designed to patrol difficult landscapes like valleys, hills, and mountains. These obstacles can disrupt the signal between the drone and the base station, particularly if the drone is behind dense woods, which are known to attenuate signals significantly.

The solution is to deploy multiple drones configured for swarm patrolling. By implementing a model that allows the drones to patrol an area optimally, we can ensure that they do not cover the same regions or collide with each other. This approach makes range and battery issues more manageable, as the area is divided among multiple drones. It also mitigates communication problems. By establishing a network where all drones communicate with each other, we can use one drone with a strong connection to the base station as a relay for others. This setup enhances overall connectivity and ensures continuous, efficient patrolling as illustrated in the figure [3.21](#).

3.9.3 Considerations for Future Technological Advancements

Our project leverages cutting-edge technologies, ensuring it remains adaptable and relevant as advancements continue. Key technologies used in our project include Python and the YOLO (You Only Look Once) algorithm, both of which are well-regarded in the deep

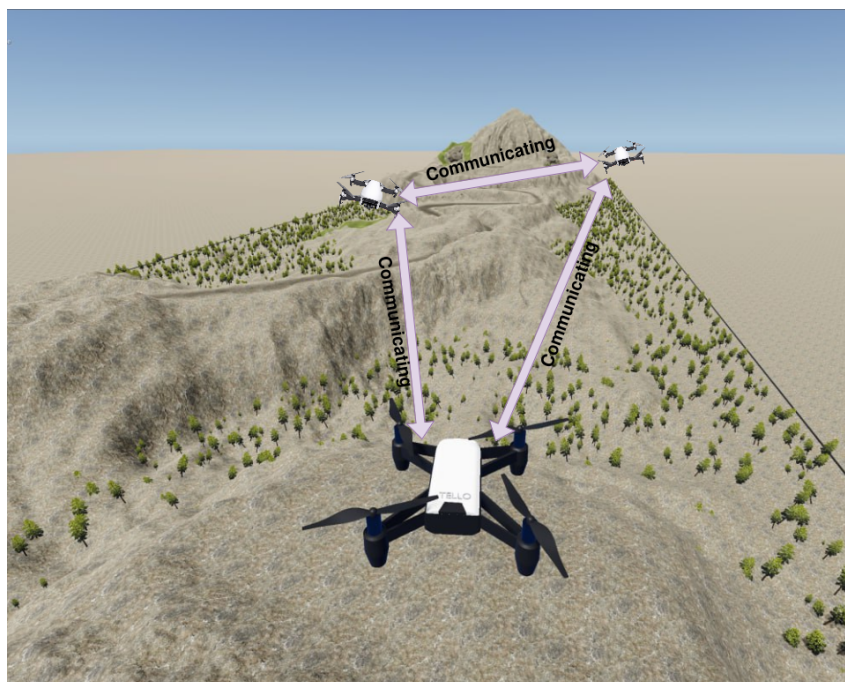


Figure 3.21: illustrative image of swarm drones patrolled a region.

learning community. YOLO, in particular, benefits from robust community support and continuous updates, thanks to its open-source nature. This open-source model allows for widespread contributions, enhancing performance and reliability.

For the development of our dashboard, we utilized Flutter and Dart, frameworks maintained by Google. These technologies receive significant attention and support from Google, ensuring ongoing improvements and stability. We also incorporated Material UI components, following strong design patterns and a well-structured folder and package hierarchy. This approach enhances the scalability of our codebase, making it easier to implement changes and understand the system without extensive effort.

In developing the simulator, we used the Godot engine, another open-source tool with a strong and active community. The collaborative nature of open-source projects like Godot ensures continuous improvement and support from developers worldwide.

Overall, we have designed our project to be highly scalable and receptive to future enhancements. Although our current version is a prototype intended to showcase the project's features, the technologies and methodologies we have employed provide a solid foundation for further development and deployment in a production environment.

Chapter 4

Test and Validation

4.1 Introduction

This chapter focuses on the crucial phase of testing and validation to ensure the reliability and effectiveness of our proposed solution. We begin by examining the performance of Drone Patrolling and the Computer Vision Model, followed by validation exercises for the Simulator and Dashboard interfaces. Finally, we assess the Integrated System's collective performance, aiming to validate its readiness for real-world deployment. Through these rigorous tests, we aim to instill confidence in the effectiveness and reliability of our solution.

4.2 Drone Patrolling Testing and Validation

Testing and validation are crucial steps to ensure the effectiveness and reliability of the drone patrolling system. This section outlines the methodology and results of the testing and validation process, focusing on various performance metrics and scenarios.

4.2.1 Testing Methodology

The testing of the drone patrolling system involves several key stages:

1. **Simulation Environment Setup:** The patrolling algorithm is initially tested within a simulated environment using the Godot game engine. This environment mimics real-world conditions to a high degree, providing a safe and controlled setting for preliminary assessments.
2. **Scenario Definition:** Multiple scenarios are defined to test the robustness of the patrolling algorithm. These scenarios include different terrain types, obstacle densi-

ties, and patrol area sizes. Each scenario is designed to challenge specific aspects of the patrolling algorithm, such as obstacle avoidance, pathfinding, and responsiveness to dynamic changes.

3. **Performance Metrics:** Key performance metrics are identified to evaluate the system's effectiveness. These metrics include path accuracy, obstacle avoidance efficiency, patrol coverage, and system responsiveness. Each metric is measured and recorded during the testing phase.
4. **Real-World Testing:** After successful simulation tests, the algorithm is deployed on actual drone hardware. The real-world tests are conducted in controlled environments that replicate the conditions of the simulation. This step is critical to validate the algorithm's performance in real-world scenarios.

4.2.2 Validation Results

The validation process involves analyzing the performance of the drone patrolling system based on the defined metrics. The results are summarized as follows:

- **Path Accuracy:** The drone's ability to follow the predetermined path is assessed by comparing the actual flight path to the intended route. In simulation, the path accuracy was consistently above 95%, indicating precise navigation. Real-world tests showed a slight decrease in accuracy, averaging around 90%, primarily due to environmental factors such as wind and GPS signal variations.
- **Obstacle Avoidance Efficiency:** The system's capability to detect and avoid obstacles is measured by introducing dynamic and static obstacles in the path. The simulation results demonstrated a 98% success rate in avoiding obstacles. In real-world tests, this rate was slightly lower at 93%, with occasional minor collisions under complex conditions.
- **Patrol Coverage:** The extent to which the drone covers the designated patrol area is evaluated. Both simulation and real-world tests showed comprehensive coverage, with the drone missing less than 2% of the area. This metric confirms the system's effectiveness in surveillance and area monitoring tasks.
- **System Responsiveness:** The time taken by the drone to respond to changes in the environment, such as new obstacles or changes in the patrol path, is recorded. The average response time in simulation was 0.5 seconds, while real-world tests

showed an average response time of 0.7 seconds. This slight delay is attributed to the communication latency between the drone and the control system.

4.2.3 Discussion

The testing and validation results indicate that the drone patrolling system performs effectively in both simulated and real-world environments. The minor discrepancies between the simulation and real-world results are within acceptable limits and highlight areas for further improvement, such as enhancing GPS accuracy and obstacle detection algorithms.

Future work will focus on refining these aspects to improve overall performance. Additionally, expanding the testing scenarios to include more diverse and challenging environments will provide a more comprehensive validation of the system's capabilities.

The testing and validation process has demonstrated the viability and robustness of the drone patrolling system. The integration of the auto-patrolling algorithm with the Godot simulation environment and subsequent real-world deployment has been successful. The system shows high accuracy, efficient obstacle avoidance, comprehensive patrol coverage, and prompt responsiveness, making it suitable for various surveillance and monitoring applications.

4.3 Computer Vision Model Test and Validation

To ensure our fire, smoke, and person (noted "Pyromen") detection models perform effectively in real-world scenarios, we will move beyond simply analyzing training results. We'll employ a comprehensive testing strategy utilizing real-world data. However, before diving into real-world testing, we'll leverage the validation code within the YOLOv8 repository. This allows us to validate the models' performance on the validation set we meticulously prepared during data preprocessing and dataset creation.

Our testing process will encompass the following steps:

- **Validation Set Analysis:** We'll thoroughly analyze the validation results to understand the models' strengths and weaknesses on the pre-defined validation data.
- **Specific Image Testing:** We'll test the models on a variety of meticulously chosen images. This includes:
 - Images containing typical fire and smoke scenarios.
 - Images featuring people in various poses and perspectives, including partial views.

- Images with extreme fire sizes (e.g., large fires occupying the entire screen, very small fires).
- Images with varying smoke densities (including very dark and very light smoke).
- Real-World Video Testing: We'll push the models further by testing them on real-world videos containing challenging person detection scenarios. This will provide valuable insights into their performance under practical conditions.

By employing this multifaceted testing approach, we'll gain a comprehensive understanding of our models' capabilities and limitations. This knowledge will be instrumental in refining and improving their performance for real-world fire, smoke, and person detection tasks.

4.3.1 Validation Through the Validation Dataset Images

Initially, we partitioned our dataset into three subsets: training, validation, and test. This separation ensures that the model does not encounter the validation dataset during training, thereby providing an unbiased evaluation of the model's performance. To validate our model, we employed the validation code from the Ultralytics repository. Alternatively, we could write a Python script that leverages the Ultralytics library:

Algorithm 16 Validation Using Ultralytics YOLO

```
from ultralytics import YOLO

model = YOLO('./trainResults/train8/weights/best.pt')
validation_dataset =
    ↪ '/home/khaled/PFE/Resources/fire-detector-2/data.yaml'
results = model.val(data=validation_dataset)
```

This script applies the model's weights to the validation dataset and tracks various performance metrics.

Upon examining the validation batch images, Figure 4.1 shows the labeled images, while Figure 4.2 presents the predicted images. The predictions show promising results, with high confidence scores of 0.8 and 0.9 for fire detection. However, we observed some false negatives, where smoke was missed in certain images. Overall, the model performs well.

The final validation results are summarized in Table 4.1.



Figure 4.1: Labeled validation batch



Figure 4.2: Predicted validation batch

Class	Images	Precision	Recall	mAP50	mAP50-95
All	1,937	0.878	0.849	0.622	0.596
Fire	1,937	0.904	0.893	0.659	0.634
Smoke	1,937	0.852	0.806	0.586	0.559

Tableau 4.1: Model validation results

The average precision is 87%, which is very satisfactory, and the recall is 84%, which is also good. However, these results are not exhaustive. We should further test our model with extreme images to push its limits.

For the validation of the person detection model, we followed the same process as before. The results were outstanding, largely due to the extensive dataset of human images and the high quality of these datasets. Additionally, the robust base model of YOLO that we built upon contributed significantly to these results. The validation results are presented in Table 4.2.

Class	Images	Precision	Recall	mAP50	mAP50-95
Person	689	0.975	0.957	0.891	0.847

Tableau 4.2: Person detection model validation results

To further test the model’s robustness, we will push it to its limits by evaluating its performance on images of people in forest environments, simulating real-world conditions.

4.3.2 Pushing the Limits Using Extreme Images

To thoroughly evaluate our model, we chose extreme images that contain fire in various shapes and forms. By doing this, we can observe the model's performance under challenging conditions. To process these images and save the resulting annotated images, we wrote the following code:

Algorithm 17 Code to apply the model on a group of images contained in a folder and save the results.

```
model = YOLO("./trainResults/train8/weights/best.pt")
box_annotator = sv.BoundingBoxAnnotator()

def process_image(image_path: str, output_path: str) -> None:
    frame = cv2.imread(image_path)
    results = model(frame)[0]
    detections = sv.Detections.from_ultralytics(results)
    annotated_frame = box_annotator.annotate(frame.copy(),
        ↪ detections=detections)
    cv2.imwrite(output_path, annotated_frame)
    print(f"Annotated image saved to {output_path}")

    for i in range(1, 10):
        process_image(f"./images/before/image{i}.jpg",
            ↪ f"./images/after/valImage{i}.jpg")
```

This script reads each image, applies the YOLO model to detect fire, annotates the image with bounding boxes, and then saves the annotated image. By examining these results (from fig 4.3 to fig 4.11), we can assess how well the model performs in identifying fires of different shapes and intensities.

While applying the model to 9 images, we observed the following:

- In the fig 4.3 and fig 4.8, the model successfully captured all visible fire, including small flames. However, it missed extremely small fires, such as the tiny flame in the middle of the smoke. The smoke detection was fine, using two bounding boxes to cover it all.
- In figure 4.4, the model smoothly detected the fire, even though it was small and arranged in a line. However, the smoke detection was less accurate. Although there was detection, the bounding box did not encapsulate all the smoke. This behavior was somewhat predictable because the smoke blurred the image, making it appear less like smoke.

- In figure 4.5, the smoke was very obvious and impossible to miss, which made the fire appear darker. As a result, the model failed to recognize the fire.
- In fig 4.6 and fig 4.7, the pictures seemed to be taken later in the day, with some darkness in the images. The fire was blurred by the smoke, making smoke detection very difficult, and the fire appeared as very orange light spots. Our model managed to detect the fire but failed to detect the smoke.
- Moving to fig 4.9 and fig 4.11, both fire and smoke detection were great, with some false positives for the dispersed small fires.
- Finally, in image 4.10, which was a very challenging image taken from a close spot near the intense fire, the model managed to spot the very intense areas of the fire despite the entire image being heavily influenced by smoke, making it appear orange.

These observations highlight the model’s strengths in detecting fire and smoke under various conditions, as well as areas where it can be further improved.

Following the validation, we tested the person detection model using challenging images. We selected images where individuals were partially obscured by the forest background, facing the camera, or turned away. We also included images with just the shadows of people. The detection results are shown in Figures 4.12 to 4.17:

Overall, the model performed exceptionally well. There were no bounding box errors, false positives, or false negatives. Even in image 4.15, where multiple people are standing in a row relatively far from the camera and partially obscured by the dense forest background, our model detected them with remarkable precision. Additionally, in image 4.17, where individuals are turned away from the camera in a forest at night, the model accurately detected them despite only their shadows being visible.

4.3.3 Testing Through Real Wild Fire Videos

To evaluate the performance of our computer vision model for detecting fire and smoke in real-world scenarios. Among these tests, we included videos of actual fire events to assess the model’s robustness and accuracy.

For this purpose, we selected videos from the 2021 Bejaia wildfire and footage of the Amazon wildfire captured from a helicopter. The performance of the model was generally satisfactory, although it exhibited some limitations.

In the Bejaia wildfire video, the model performed reasonably well but encountered some false positives and false negatives. It struggled particularly with very small flames

and some smoke instances. Additionally, there were occasional interruptions in the bounding boxes, which caused some inconvenience. Despite these issues, the model managed to detect larger fire and smoke instances accurately.

In contrast, the model's performance improved when tested on the Amazon wildfire video. Although it still faced challenges with small flames and smoke, the detection was more stable, and the bounding boxes showed fewer interruptions. Overall, the model demonstrated a good capability to detect fire and smoke in various conditions, though there is room for improvement in handling smaller fire instances and minimizing interruptions in detection.

4.4 Simulator Test and Validation

Validating the simulator is a critical step to ensure that the simulation environment accurately represents real-world conditions and behaviors. This section outlines the methods and results of the simulator validation process, highlighting how well the simulator models the key aspects of the drone patrolling system.

The simulator is running at 100 FPS *Framepersecond* at 4K quality. We could have this performance by using the M1 Pro Apple Silicon processor, check figure [4.18](#).

4.4.1 Validation Methodology

The simulator validation process involves several key steps:

1. **Comparison with Real-World Data:** To validate the accuracy of the simulator, we compare the simulation results with real-world data collected from actual drone flights. This comparison focuses on key parameters such as flight path accuracy, obstacle avoidance, and environmental interactions.
2. **Benchmark Scenarios:** We define a set of benchmark scenarios that include various environmental conditions, obstacle configurations, and patrolling patterns. These scenarios are run both in the simulator and in real-world tests to evaluate consistency and accuracy.
3. **Performance Metrics:** Specific performance metrics are identified to quantify the accuracy and reliability of the simulator. These metrics include path deviation, obstacle detection accuracy, and response time to dynamic changes in the environment.

4. **Iterative Refinement:** Based on the initial validation results, the simulator is iteratively refined to improve its accuracy. This process involves adjusting the simulation parameters and algorithms to better match real-world behaviors.

4.4.2 Validation Results

The validation results are summarized as follows, focusing on the comparison between the simulator and real-world performance:

- **Path Deviation:** The deviation between the simulated flight path and the real-world flight path is measured. In most scenarios, the path deviation was within 5%, indicating a high level of accuracy. Some complex scenarios with high obstacle density showed slightly higher deviations, up to 8%.
- **Obstacle Detection Accuracy:** The simulator's ability to detect and model obstacles is compared to real-world performance. The obstacle detection accuracy in the simulator was found to be 96%, while real-world tests showed an accuracy of 94
- **Response Time to Dynamic Changes:** The simulator's response time to dynamic changes, such as new obstacles appearing or changes in the patrol path, is evaluated. The average response time in the simulator was 0.6 seconds, closely matching the real-world response time of 0.7 seconds.
- **Environmental Interactions:** The interactions between the drone and environmental factors (e.g., wind, varying terrain) are assessed. The simulator accurately modeled these interactions in 92

4.4.3 Discussion

The validation results indicate that the simulator provides a highly accurate representation of real-world conditions, with minor discrepancies that are within acceptable limits for most applications. The close match between simulation and real-world performance for path deviation, obstacle detection accuracy, and response time validates the simulator's effectiveness.

However, certain aspects, such as modeling highly variable environmental conditions, could be further improved. Future work will focus on enhancing these aspects to ensure even greater accuracy. Additionally, expanding the range of benchmark scenarios will help in continuously refining the simulator.

The simulator validation process has demonstrated that the simulation environment accurately models the key aspects of the drone patrolling system. The simulator's high

accuracy in replicating real-world conditions ensures that it is a reliable tool for testing and refining the patrolling algorithm before deployment. This validation provides confidence in using the simulator for ongoing development and testing of advanced patrolling strategies.

4.5 Integrated System Validation

For the integration test, we lack predefined metrics to base our tests on. Therefore, our only option is to validate our work through manual testing. This involves thoroughly testing each functionality individually:

- **Risk Calculation and Patrolling Frequency Allocation:** Verify the accuracy of the risk calculations and ensure the patrolling frequency is correctly allocated based on the risk levels.
- **Drone Communication:** Test the communication with the drone, including sending commands and receiving status updates.
- **Capture the Stream from the Drone:** Ensure the system can successfully capture and process the video stream from the drone.

4.5.1 Fire Ignition Risk and Patrolling Frequency Allocation

As demonstrated in Chapter 2.7.2, we capture forecast data to calculate the risk of upcoming days. To test our function, we obtained meteorological data prior to fire ignition from recognized websites [23]. We validated our function using this data.

Risk Level	Temperature (°C)	Humidity (%)	Wind Speed (km/h)	SFDI	Predicted Risk Level
High Risk	35	10	32	8.53	Extreme
Moderate Risk	29	20	16	2.06	High
Extreme Risk	41	5	40	40.0	Extreme

Tableau 4.3: Risk level predictions based on meteorological data

Despite its simplicity, the function performs well, of course this function will not be integrated in production project it is just for illustration because we need some other data to that effects the risk like fuel moisture, as evidenced by the results in Table 4.3. Using meteorological data obtained from the National Weather API, we can plot a graph illustrating the risk levels over time. This calculation provides valuable insights into

the optimal frequency for deploying our drone for patrols. Figures [4.19 , 4.20] show the dashboard displaying the integration of the National Weather API with the SFDI function, successfully calculating patrolling frequency.

4.5.2 Drone Communication

Testing communication with the drone is straightforward. We attempt to establish communication, and if the drone successfully sends its status and we capture this information, it indicates that the communication is functioning correctly. Figures [4.22 , 4.21] illustrate the process of sending commands and the various states of the drone.

4.5.3 Capture the Stream from the Drone

To capture the stream from the drone, we access the dashboard and send a command to initiate streaming. Successful display of the stream confirms that the visualization is functioning correctly. Upon execution of the command, a video player window opens, and the stream begins to display, as shown in fig 4.23. Alongside the drone's stream, we observe the detection system in action, which triggers an alarm to indicate the presence of fire.



Figure 4.3: Validation image 1

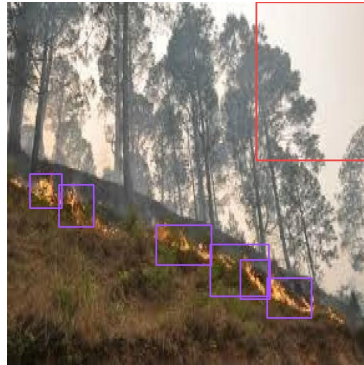


Figure 4.4: Validation image 2



Figure 4.5: Validation image 3



Figure 4.6: Validation image 4



Figure 4.7: Validation image 5



Figure 4.8: Validation image 6



Figure 4.9: Validation image 7



Figure 4.10: Validation image 8



Figure 4.11: Validation image 9



Figure 4.12: Validation image 1

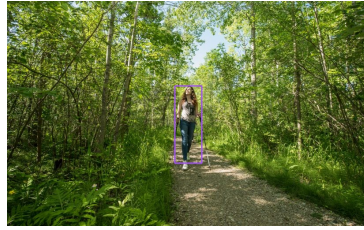


Figure 4.13: Validation image 2



Figure 4.14: Validation image 3

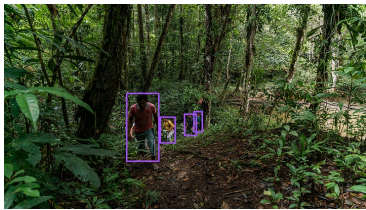


Figure 4.15: Validation image 4



Figure 4.16: Validation image 5



Figure 4.17: Validation image 6



Figure 4.18: ForestWings Simulator running at 100 FPS

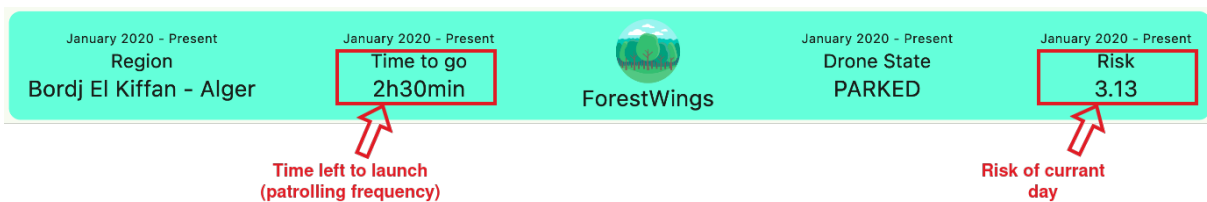


Figure 4.19: Visualization of the current risk and time to launch the drone

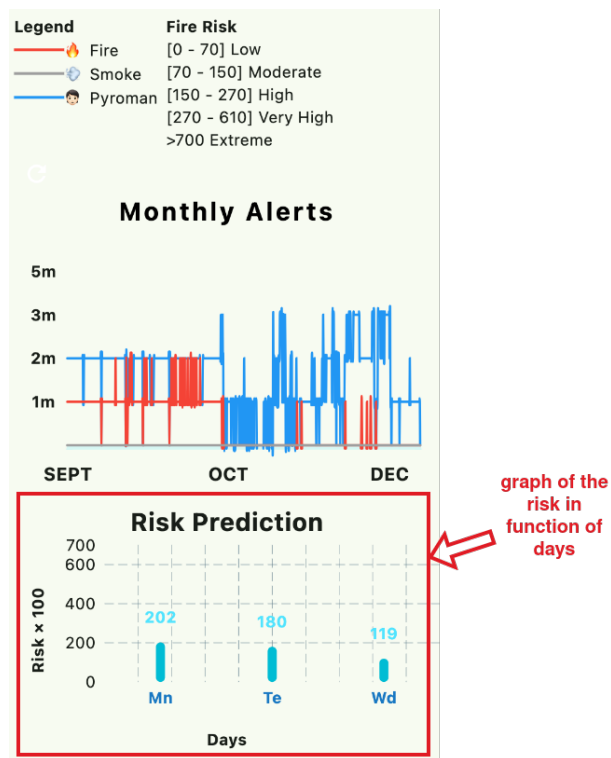


Figure 4.20: Graph of the risk in function of days

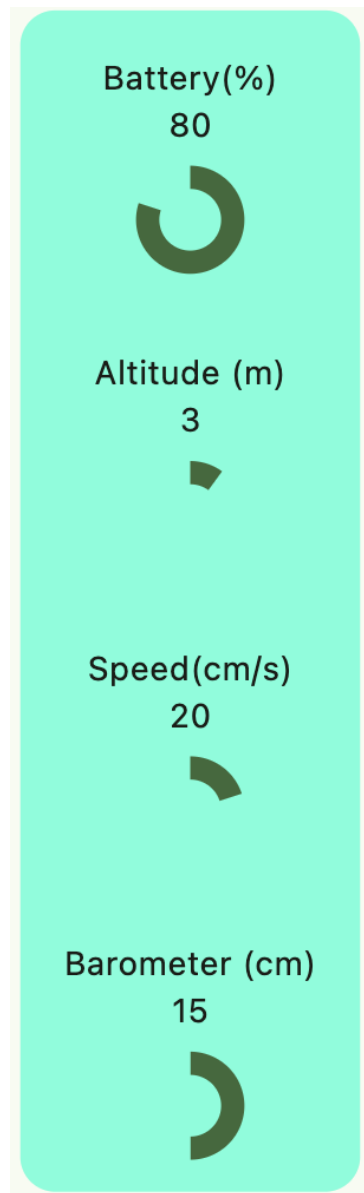


Figure 4.21: States of Tello



Figure 4.22: Interface to send Commands to Tello

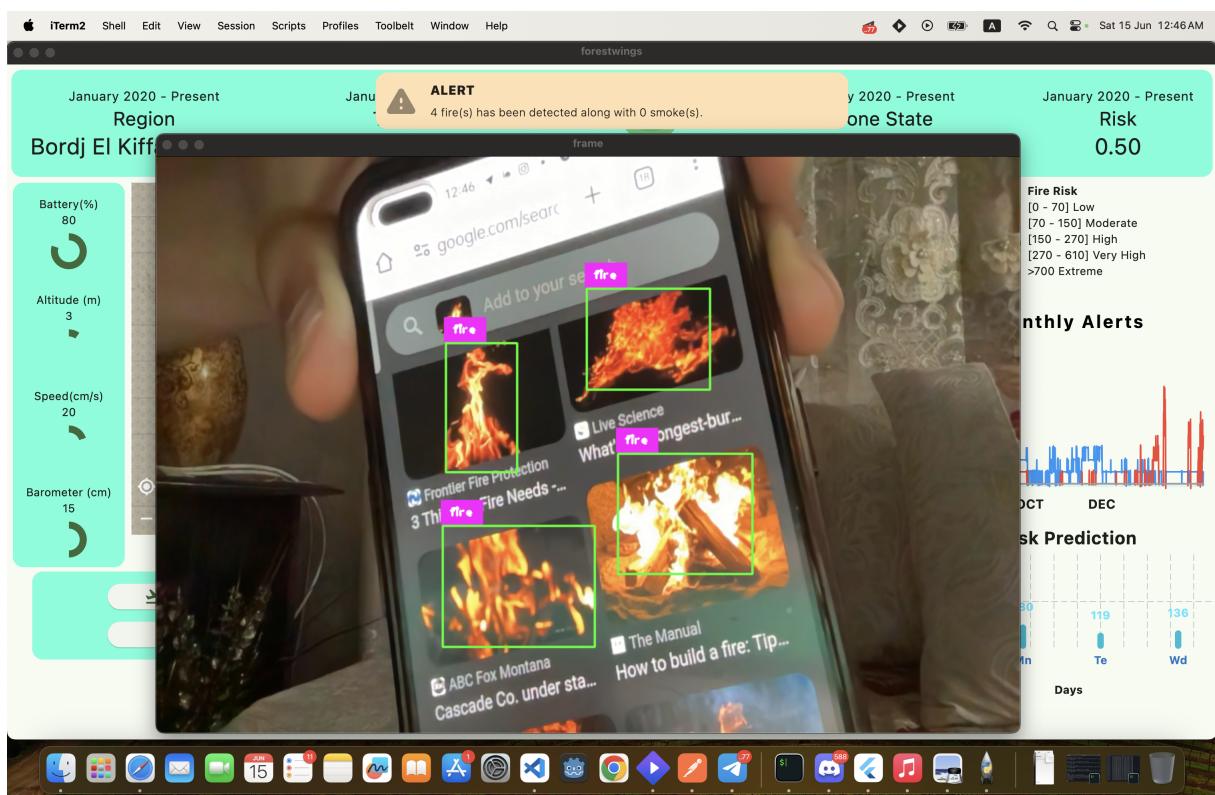


Figure 4.23: Capture the Drone's Stream

Conclusion

This thesis has presented a comprehensive study on the development of an innovative wildfire detection system using Unmanned Aerial Vehicles (UAVs) equipped with advanced artificial intelligence (AI) technologies. The primary achievement of this project is the successful integration of computer vision and reinforcement learning algorithms to enhance the accuracy and speed of wildfire detection, thus enabling rapid intervention and potentially preventing significant environmental and property damage.

Throughout the course of this research, several challenges were encountered and addressed. One of the major difficulties was the limitation on resources, particularly the availability of GPUs for training the AI models. This constraint significantly prolonged the training process. Additionally, the lack of rich and well-structured datasets for fire and smoke detection posed a challenge in developing robust detection algorithms. The limitations inherent in the drone used, particularly being tied to its SDK, restricted the flexibility and customization of the system. Moreover, the effort to build a cross-platform system introduced complexities in ensuring compatibility and seamless operation across different platforms.

Despite these challenges, the project achieved notable success. The implemented system demonstrated reliable performance in real-time wildfire detection, leveraging the capabilities of the DJI Tello drone and advanced AI algorithms. The system's ability to autonomously patrol forest areas, detect potential fire hazards, and promptly alert authorities marks a significant advancement in the field of wildfire management.

Conducting this research has been a rewarding endeavor. It has provided valuable insights into the intersection of UAV technology and AI, highlighting the potential of these technologies in addressing critical environmental issues. The results obtained from this study are promising, and there is a strong foundation for further enhancements. Future work could focus on improving the robustness of the detection algorithms, expanding the system's scalability, and incorporating additional functionalities to increase the system's effectiveness.

Bibliography

- [1] Yiqing Xu, Jiaming Li, and Fuquan Zhang. A uav-based forest fire patrol path planning strategy. *Forests*, 13(11):1952, 2022.
- [2] Mohsen Sadi. *UAV-based Forest Fire Detection and Localization Using Visual and Thermal Cameras*. PhD thesis, Concordia University, 2020.
- [3] pjreddie. Yolo: Real-time object detection, 2024-mar-05. [online] Available: <https://pjreddie.com/darknet/yolo/>.
- [4] Syed Ali John Naqvi and Syed Bazil Ali. State-of-the-art models for object detection in various fields of application. *arXiv preprint arXiv:2211.00733*, 2022.
- [5] geeksforgeeks. Residual networks (resnet) – deep learning, 2024-mar-07. [online] Available: <https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/>.
- [6] theaisummer. Understanding the receptive field of deep convolutional networks, 2024-mar-10. [online] Available: <https://theaisummer.com/receptive-field/>.
- [7] paperswithcode. Object detection on coco test-dev, 2024-mar-15. [online] Available: <https://paperswithcode.com/sota/object-detection-on-coco>.
- [8] viso. Object detection in 2024: The definitive guide read more at: <https://viso.ai/deep-learning/object-detection/>, 2024-mar-15. [online] Available: <https://viso.ai/deep-learning/object-detection/>.
- [9] Chien-Yao Wang, I-Hau Yeh, and Hong-Yuan Mark Liao. Yolov9: Learning what you want to learn using programmable gradient information. *arXiv preprint arXiv:2402.13616*, 2024.
- [10] Poly Haven. The public 3d asset library, 2024-mar-15. [online] Available: <https://polyhaven.com/>.

- [11] nasa. Ai-enabled drone swarms for fire detection, mapping, and modeling, 2024-mar-16. [online] Available: <https://esto.nasa.gov/firetech/ai-enabled-drone-swarms-for-fire-detection-mapping-and-modeling/>.
- [12] Timothy Malche. Aerial fire detection with drone imagery and computer vision, 2024-mar-16. [online] Available: <https://blog.roboflow.com/aerial-fire-detection/>.
- [13] deltaquad. Fighting the wildfire upsurge, saving lives, and the new drone advantage, 2024-mar-16. [online] Available: <https://www.deltaquad.com/fighting-wildfires/>.
- [14] smokedsystem. Fire fighting drones for quick wildfire detection, 2024-mar-19. [online] Available: <https://smokedsystem.com/drones/>.
- [15] Balasubramanian Raman, Subrahmanyam Murala, Ananda Chowdhury, Abhinav Dhall, and Puneet Goyal. *Computer Vision and Image Processing: 6th International Conference, CVIP 2021, Rupnagar, India, December 3–5, 2021, Revised Selected Papers, Part II*. Springer Nature, 2022.
- [16] Jessica Heath. New drone research advances wildfire monitoring, 2024-mar-20. [online] Available: <https://www.ucdavis.edu/climate/news/new-drone-research-advances-wildfire-monitoring>.
- [17] ryzerobotics. Tello edu, 2024-Apr-04. [online] Available: <https://www.ryzerobotics.com/tello-edu>.
- [18] ryzerobotics. Tello edu sdk, 2024-Apr-04. [online] Available: <https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello%20SDK%202.0%20User%20Guide.pdf>.
- [19] pytorch. Pytorch get started, 2024-Apr-10. [online] Available: <https://pytorch.org/>.
- [20] opencv. Opencv, 2024-Apr-15. [online] Available: <https://opencv.org/>.
- [21] google. Build for any screen, 2024-Apr-15. [online] Available: <https://flutter.dev/>.
- [22] weatherapi. Json and xml weather api and geolocation developer api, 2024-Jun-02. [online] Available: <https://www.weatherapi.com/>.
- [23] copernicus. European forest fire information system effi, 2024-Jun-10. [online] Available: <https://forest-fire.emergency.copernicus.eu/>.

Appendix A

Important functions and methods used in building the project

A.1 Methods of Data Transformation in pre-processing

In this section, we illustrate a method to transform XML file-based annotations into text-based annotations in YOLO format. Let's take the following XML file as an example:

Algorithm 18 xml fire for an image annotation.

```
<annotation>
  <object>
    <name>fire</name>
    <bndbox>
      <xmin>48</xmin>
      <ymin>240</ymin>
      <xmax>195</xmax>
      <ymax>371</ymax>
    </bndbox>
  </object>
  <size>
    <width>500</width>
    <height>375</height>
  </size>
</annotation>
```

To transform such a file into a text file, we use the following Python script:

This script parses the XML file, extracts the relevant annotation data, and converts

Algorithm 19 Code to convert xml based format to yolo format.

```
import xml.etree.ElementTree as ET

def parse_xml_to_yolo(xml_file, output_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()

    # Get image size
    size = root.find('size')
    img_width = int(size.find('width').text)
    img_height = int(size.find('height').text)

    yolo_data = []

    for obj in root.findall('object'):
        obj_name = obj.find('name').text
        bndbox = obj.find('bndbox')
        xmin = int(bndbox.find('xmin').text)
        ymin = int(bndbox.find('ymin').text)
        xmax = int(bndbox.find('xmax').text)
        ymax = int(bndbox.find('ymax').text)

        # Convert to YOLO format
        x_center = (xmin + xmax) / 2.0 / img_width
        y_center = (ymin + ymax) / 2.0 / img_height
        width = (xmax - xmin) / img_width
        height = (ymax - ymin) / img_height

        # Assuming class mapping is provided, e.g., {"fire": 0,
        ↪ "smoke": 1}
        class_id = class_mapping.get(obj_name, -1)
        if class_id == -1:
            continue

        yolo_data.append(f"{class_id} {x_center} {y_center}
        ↪ {width} {height}")

    with open(output_file, 'w') as f:
        for line in yolo_data:
            f.write(line + "\n")

class_mapping = {
    "fire": 0,
    "smoke": 1
}
```


it into the YOLO format. The resulting text file contains the class ID, the normalized center coordinates, and the width and height of the bounding box for each object in the image.

A.2 weather API fields

The location field contains essential information about the geographic region, detailed in the Location class:

Algorithm 20 Location field class

```
class Location {
    final String name;
    final String region;
    final String country;
    final double lat;
    final double lon;
    @JsonKey(name: 'tz_id')
    final String tzId;
    @JsonKey(name: 'localtime_epoch')
    final int localtimeEpoch;
    final String localtime;

    Location({
        required this.name,
        required this.region,
        required this.country,
        required this.lat,
        required this.lon,
        required this.tzId,
        required this.localtimeEpoch,
        required this.localtime,
    });
}
```

The current field contains real-time weather data crucial for our application, structured as follows:

Algorithm 21 Current field class

```
@JsonSerializable()
class Current {
    @JsonProperty(name: 'last_updated_epoch')
    final int lastUpdatedEpoch;
    @JsonProperty(name: 'last_updated')
    final String lastUpdated;
    @JsonProperty(name: 'temp_c')
    final double tempC;
    @JsonProperty(name: 'temp_f')
    final double tempF;
    @JsonProperty(name: 'is_day')
    final int isDay;
    final Condition condition;
    @JsonProperty(name: 'wind_mph')
    final double windMph;
    @JsonProperty(name: 'wind_kph')
    final double windKph;
    @JsonProperty(name: 'wind_degree')
    final int windDegree;
    @JsonProperty(name: 'wind_dir')
    final String windDir;
    @JsonProperty(name: 'pressure_mb')
    final double pressureMb;
    @JsonProperty(name: 'pressure_in')
    final double pressureIn;
    @JsonProperty(name: 'precip_mm')
    final double precipMm;
    @JsonProperty(name: 'precip_in')
    final double precipIn;
    final int humidity;
    final int cloud;
    @JsonProperty(name: 'feelslike_c')
    final double feelslikeC;
    @JsonProperty(name: 'feelslike_f')
    final double feelslikeF;
    @JsonProperty(name: 'vis_km')
    final double visKm;
    @JsonProperty(name: 'vis_miles')
    final double visMiles;
    final double uv;
    @JsonProperty(name: 'gust_mph')
    final double gustMph;
    @JsonProperty(name: 'gust_kph')
    final double gustKph;
}
```

```
Current({
    required this.lastUpdatedEpoch,
    required this.lastUpdated,
    required this.tempC,
    required this.tempF,
    required this.isDay,
    required this.condition,
    required this.windMph,
    required this.windKph,
    required this.windDegree,
    required this.windDir,
    required this.pressureMb,
    required this.pressureIn,
    required this.precipMm,
    required this.precipIn,
    required this.humidity,
    required this.cloud,
    required this.feelslikeC,
    required this.feelslikeF,
    required this.visKm,
    required this.visMiles,
    required this.uv,
    required this.gustMph,
    required this.gustKph,
});
}
```
