UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

UAV-Based Wildfire Detection using RGB and Thermal Cameras

Diploma Thesis

Vasileios Anagnostopoulos

Supervisor: Christos Antonopoulos

July 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανίχνευση Δασικών Πυρκαγιών Βασισμένη σε ΜηΕΑ, Αξιοποιώντας RGB και Θερμική Κάμερα

Διπλωματική Εργασία

Βασίλειος Αναγνωστόπουλος

Επιβλέπων/πουσα: Χρήστος Αντωνόπουλος

Ιούλιος 2025

Approved by the Examination Committee:

Supervisor **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to express my sincere gratitude to Professor Christos Antonopoulos for his guidance and support throughout the duration of this thesis. I am especially thankful for his willingness to embrace my proposed topic and for giving me the opportunity to bring my idea to life under his supervision. His insightful feedback and constructive suggestions were invaluable during both the research and implementation phases.

I am also grateful to the faculty of the Department of Electrical and Computer Engineering at the University of Thessaly for creating an environment that fosters curiosity, critical thinking, and academic growth. I would like to thank my fellow students and colleagues who contributed to this journey through thoughtful discussions, technical advice, and moral support.

Finally, I would like to express my deepest gratitude to my family and close friends, who have supported me wholeheartedly throughout the five-year journey of my studies. My family has been a pillar of strength, offering endless encouragement, emotional support, and unwavering belief in my potential, even during the most difficult moments. Their sacrifices, patience, and love have been instrumental in helping me reach this milestone. At the same time, my close friends have been by my side through late nights, stressful deadlines, and moments of doubt, always ready to listen, motivate, and remind me of my goals. This accomplishment is as much theirs as it is mine, and I am truly grateful to have had such incredible people beside me every step of the way.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Vasileios Anagnostopoulos

Diploma Thesis

UAV-Based Wildfire Detection using RGB and Thermal Cameras

Vasileios Anagnostopoulos

Abstract

Wildfires are a significant environmental threat, causing severe ecological and economic damage while endangering human lives. This thesis presents an autonomous UAV-based wildfire detection system that leverages both RGB and thermal cameras to improve detection performance under varying visibility and environmental conditions. RGB imagery is used to detect fire and smoke, while thermal imagery enables the detection of fire, humans, and animals, extending the system's capabilities in low-visibility and heat-based scenarios.

The system is implemented within a simulated environment developed in Unreal Engine and is managed through a ROS 2-based processing pipeline. Two custom datasets were created from different sources to train and evaluate multiple deep learning-based object detection models. After comparative analysis, the best-performing models were selected and integrated into the final system. Image registration is applied to align thermal and RGB frames, while a fusion module combines the synchronized outputs of both detectors to produce consistent and reliable multi-modal results. Additionally, object tracking is incorporated to maintain detection consistency over time and enable more robust alert generation.

The proposed system demonstrates the feasibility of UAV-based wildfire monitoring and highlights the potential of combining multi-sensor data, image registration, and intelligent detection for enhanced situational awareness in simulated wildfire environments. The modular and simulation-driven approach lays a strong foundation for potential real-world deployment in remote and high-risk areas.

Keywords: UAV, wildfire detection, RGB camera, thermal camera, ROS 2, Unreal Engine, object detection, deep learning, object tracking, image registration, multi-modal fusion

Διπλωματική Εργασία

Ανίχνευση Δασικών Πυρκαγιών Βασισμένη σε ΜηΕΑ, Αξιοποιώντας RGB και Θερμική Κάμερα

Βασίλειος Αναγνωστόπουλος

Περίληψη

Οι δασικές πυρκαγιές αποτελούν μια σοβαρή περιβαλλοντική απειλή, προκαλώντας σημαντικές οικολογικές και οικονομικές ζημιές, ενώ ταυτόχρονα θέτουν σε κίνδυνο ανθρώπινες ζωές. Η παρούσα διπλωματική εργασία παρουσιάζει ένα αυτόνομο σύστημα ανίχνευσης δασικών πυρκαγιών με χρήση μη επανδρωμένου αεροσκάφους (UAV), το οποίο αξιοποιεί κάμερες RGB και θερμικές για τη βελτίωση της ακρίβειας ανίχνευσης υπό διαφορετικές συνθήκες ορατότητας και περιβάλλοντος. Η κάμερα RGB χρησιμοποιείται για την ανίχνευση φωτιάς και καπνού, ενώ η θερμική κάμερα επιτρέπει την ανίχνευση φωτιάς, ανθρώπων και ζώων.

Το σύστημα έχει υλοποιηθεί σε προσομοιωμένο περιβάλλον που αναπτύχθηκε στο Unreal Engine και συντονίζεται μέσω αρχιτεκτονικής ROS 2. Δημιουργήθηκαν δύο προσαρμοσμένα σύνολα δεδομένων από διαφορετικές πηγές για την εκπαίδευση και αξιολόγηση διαφόρων μοντέλων εντοπισμού αντικειμένων με χρήση βαθιάς μάθησης. Εφαρμόζεται ευθυγράμμιση εικόνας μεταξύ RGB και θερμικής κάμερας, ενώ η συγχώνευση των αποτελεσμάτων παράγει αξιόπιστες πολυτροπικές ανιχνεύσεις. Η ενσωμάτωση παρακολούθησης αντικειμένων βελτιώνει τη χρονική συνέπεια και την ποιότητα των ειδοποιήσεων.

Το προτεινόμενο σύστημα αποδεικνύει την αποτελεσματικότητα της αξιοποίησης UAV για την παρακολούθηση δασικών πυρκαγιών και αναδεικνύει τις δυνατότητες του συνδυασμού δεδομένων από πολλαπλούς αισθητήρες, ευθυγράμμισης εικόνας και ευφυούς ανίχνευσης για ενισχυμένη επιχειρησιακή επίγνωση σε προσομοιωμένα περιβάλλοντα πυρκαγιάς. Η αρθρωτή, βασισμένη σε προσομοίωση προσέγγιση, θέτει τις βάσεις για μελλοντική εφαρμογή σε πραγματικά σενάρια σε απομακρυσμένες ή υψηλού κινδύνου περιοχές.

Λέξεις-Κλειδιά: UAV, ανίχνευση πυρκαγιάς, κάμερα RGB, θερμική κάμερα, ROS 2, Unreal Engine, εντοπισμός αντικειμένων, βαθιά μάθηση, παρακολούθηση αντικειμένων, ευθυγράμμιση εικόνας, συγχώνευση δεδομένων

Table of contents

List of figures

List of tables

Abbreviations

i.e.	id est
e.g.	exempli gratia
etc.	et cetera
UAV	Unmanned Aerial Vehicle
ML	Machine Learning
AI	Artificial Intelligence
PCA	Principal Component Analysis
SVD	Singular Value Decomposition
GPU	Graphics Processing Unit
CNN	Convolutional Neural Network
ANN	Artificial Neural Network
FC	Fully Connected
FPN	Feature Pyramid Network
PANet	Path Aggregation Network
BiFPN	Bidirectional Feature Pyramid Network
RPN	Region Proposal Network
ROI	Region Of Interest
NMS	Non-Maximum Suppression
SOT	Single Object Tracking
MOT	Multiple Object Tracking
CPU	Central Processing Unit
GPS	Global Positioning System
LiDAR	Light Detection And Ranging
IR	Infrared
NIR	Near-Infrared

SWIR	Small-Wavelength Infrared
MWIR	Mid-Wavelength Infrared
LWIR	Long-Wavelength Infrared
FIR	Far-Infrared
ROS	Robot Operating System
UE	Unreal Engine
DDS	Data Distribution Service
QoS	Quality of Service
RTOS	Real-Time Operating System
CLAHE	Contrast Limited Adaptive Histogram Equalization
YOLO	You Only Look Once
YOLOv8	You Only Look Once version 8
SSD	Single Shot Detector
R-CNN	Region-based Convolutional Neural Network
ResNet	Residual Network
CUDA	Compute Unified Device Architecture
RAM	Random Access Memory
mAP	mean Average Precision
IoU	Intersection over Union
FPS	Frames Per Second
GMC	Global Motion Compensation
CMC	Camera Motion Compensation
KF	Kalman Filter
ReID	Re-Identification
DeepSORT	Deep Simple Online and Realtime Tracking
BoT-SORT	Bag of Tricks - Simple Online and Realtime Tracking

Chapter 1

Introduction

The natural disaster of wildfires has emerged as one of the most damaging events that climate change worsens by escalating both their occurrence and intensity. Wildfires inflict permanent damage to natural environments while creating dangers to human survival and causing substantial financial harm. The key to reducing their destruction lies in rapid detection followed by an immediate emergency response. Existing satellite-based detection methods, together with ground-based patrols and fire watchtowers, have performance restrictions due to their limited resolution capacity, delayed response times, and insufficient geographic range. Monitoring operations face the most significant restrictions in areas that are both distant and hard to reach because these locations often lack proper surveillance systems along with communication networks.

Unmanned Aerial Vehicles (UAVs) are considered autonomous systems when combined with edge computing and deep learning technologies for environmental monitoring. Through their ability to move rapidly across extensive territories, UAVs achieve complete flexibility and speed, in addition to enabling real-time data processing on board, which eliminates the need for continuous server communication. The use of multi-modal sensing technologies which include RGB and thermal cameras along with UAVs enables accurate detection during adverse conditions, including poor visibility and heavy smoke together with dense vegetation.

The fusion of UAV technology with multisensor systems and AI detection algorithms presents a robust solution to improve wildfire detection capabilities. However, challenges remain in integrating and synchronizing data from different modalities, designing robust real-time inference pipelines, and ensuring effective deployment in complex environments.

1.1 Thesis Objective

This thesis presents the design, development, and evaluation of a UAV-based wildfire detection system that combines RGB and thermal imaging with deep learning techniques for real-time object detection and alerting. The system can be used in a virtual setup that has been generated in Unreal Engine and all processes are controlled through ROS 2 enhancing the modularity of each module according to perception, decision-making, and visualization factors.

RGB imagery is used to detect visible signs of fire and smoke, while thermal imaging enables the identification of heat signatures of fire, humans, and animals. Two custom datasets were developed, one for the RGB domain and the other for the thermal domain, to train and test the detection models, which were synthesized from real and virtual images.

In addition to detecting victims in disaster scenarios, the human class also has the functionality of identifying potential criminal activities such as arson. The animal class was also included to help the models learn visual and thermal dissimilarities between humans and animals, allowing the models to discern these two classes in a better manner. These two classes are exclusively part of the thermal modality, as detecting people or animals from the RGB images of a UAV at a height of about 100 meters above forest areas is not only impossible, but also unlikely due to the scale and visual occlusion problems.

The evaluation process examined a set of object detection models based on deep learning, contributing to the identification of the best performing models, which were selected based on the detection accuracy and their real-time inference performance as the choosing criteria. The final selected models were optimized with TensorRT and performance was compared with different precisions such as FP32, FP16, and INT8 levels before deployment.

The image registration process used a homography technique to spatially align the thermal camera data with RGB frames from the two modalities. The system used a data fusion module to synchronize the prediction outputs of both RGB and thermal detectors and create unified detection results. Regarding the issue of many alerts being received at every step, an object tracking mechanism was developed and embedded to accomplish two things: to trace the position of the identified objects over time and also to enable better alerting based on object IDs rather than frame-level detections.

Additionally, each detection is equipped with GPS metadata that represents the geographical positions and generates alerts in the UAV's monitoring pipeline. Despite the present im-

plementation focusing on internal logging and alerting, this architecture paves the way for future enhancements, such as direct communication with emergency response services or adaptive UAV flight control based on detection feedback.

1.1.1 Significance and Contributions

The system developed in this thesis demonstrates the feasibility of deploying intelligent wildfire detection systems using UAVs equipped with multimodal cameras and onboard deep learning inference. The key contributions of this thesis are as follows:

- Construction of two custom datasets with labeled fire, smoke, human and animal instances across RGB and thermal imagery.
- Evaluation and comparison of multiple state-of-the-art object detection models in each modality, based on both detection accuracy and performance metrics using NVIDIA RTX 4090, leading to the selection of the best models for deployment.
- Optimization of two best detection models using TensorRT with FP16 and INT8 precision, and performance evaluation on both Jetson Orin Nano and NVIDIA RTX 4090.
- Implementation of a simulation-based development and testing framework using Unreal Engine and ROS 2 for wildfire detection experiments.
- Development of an image registration pipeline to align thermal and RGB camera outputs, ensuring accurate fusion of detection results across modalities.
- Integration of object tracking across frames to reduce redundant alerts and enable persistent monitoring of fire, smoke, human, and animal-related entities.
- Incorporation of GPS metadata with detections, enabling spatial awareness and geolocated alerting within the system.
- Visualization and logging of fused and tracked detections for real-time monitoring and offline analysis in wildfire detection scenarios.

This thesis highlights how multimodal sensing and real-time AI processing can enhance early wildfire detection capabilities, especially in remote or high-risk environments where traditional surveillance methods are less effective.

1.2 Thesis Structure

The structure of this thesis is organized into seven chapters plus the Introduction chapter, each contributing to understanding, development, and evaluation of the proposed wildfire detection system. A brief overview of each chapter is provided below:

- **Chapter 2: General Background** — Provides fundamental knowledge required for this thesis on machine and deep learning, computer vision, UAVs, sensors, simulation environment and ROS 2.
- **Chapter 3: Related Work** — Reviews existing approaches to wildfire detection based on UAVs, sensor fusion techniques, setup specifications, and prior work in developing real-world scenarios with UAVs.
- **Chapter 4: ML-Based Wildfire Detection** — Describes the process of dataset generation, selection, training, and comparison of detection models, optimizations of deployed models.
- **Chapter 5: System Implementation** — Details the implementation of the system in ROS 2, including detection result synchronization, inference pipelines, data preprocessing, alignment of RGB and thermal camera streams, object tracking after detection, and the visualization and alerting of fused, tracked detection results.
- **Chapter 6: Simulation Environment Implementation** — Details the implementation of the environment in Unreal Engine, covering the creation of a realistic simulation environment with forests, fires, smoke, humans and animals. It also describes the integration of a custom thermal camera, the use of the AirSim plugin, and how the connection between Unreal Engine and AirSim is established in Linux.
- **Chapter 7: System Evaluation** — Features a real-time evaluation that demonstrates the ROS 2-based system operating within Unreal Engine through the AirSim plugin, effectively simulating a real-world deployment. It also illustrates how all components interact seamlessly, showcasing the system as a fully integrated and functional prototype. Additionally, a video demonstration showcasing the actual testing of the developed system is available at *YouTube – AirSim and ROS 2 System Simulation*. The complete source code for the project can be accessed through the *GitHub Repository*.

- **Chapter 8: Conclusions and Future Work** — Summarizes the main findings of the thesis and proposes directions for further research and optimizations.

Chapter 2

General Background

2.1 Machine Learning

Machine Learning (ML) [1] is a branch of artificial intelligence (AI) that is focused on creating algorithms that can, without the need for direct programming commands, extract knowledge from the data to be able to improve the performance of the operating system. Machine learning systems, which are distinct from regular programming methods, have the capacity to recognize patterns and decision-making rules from the sample data themselves.

At the heart of ML is the idea of learning from data. By exposing a machine learning model to a set of data and a task, the system will be able to pick out the statistical patterns and the interdependence between pieces of data that will allow it to offer predictions or even decisions when faced with new, unsoiled input. For instance, instead of a sample algorithm that was given the information of the visual aspects of smoke and fire, an image classification algorithm would learn the differences between the two respective objects by inspecting the thousands of labeled images and pointing out the visual characteristics that make them stand apart.

The capacity to comprehend and draw out relevant information from the data has helped ML acquire immense power and has enabled its deployment in a wide range of real-world applications, such as image and speech recognition, recommendations systems, natural language processing, autonomous vehicles, predictive maintenance, and the prevention of fraud. These systems, as a rule, grow better with the increase in the amount of data; they adapt and improve their knowledge to a significant extent over time by the exposure to the data.

In general, machine learning methods are often presented in three main categories: su-

pervised learning, unsupervised learning, and reinforcement learning. The design of each of these approaches allows for different types of problems to be addressed and various levels of guidance required from the data. For example, in the field of computer vision, apart from other areas, supervised learning has been the standard application for tasks such as object detection and image classification.

2.1.1 Supervised Learning

Supervised learning [1] is the most dominant and well-understood method of machine learning. In this setting, the model learns from a dataset that contains pairs of input and output. The goal is to help the model build a mapping (function) from the inputs to the outputs such as the conversion of a photo depicting smoke into a label like “fire” or “no fire”.

When the system is trained, it adjusts its internal parameters for the purpose of minimizing the error between its predictions and real outputs. Normally this is achieved through some optimization techniques, e.g., using the gradient descent algorithm. After the training phase is over, the system can utilize the gained experience to give predictions for never-seen-before data points.

Generally, there are two types of Supervised learning tasks, and these are: classification and regression. Classification is the process of predicting classes, thus, the examples would include telling whether a particular photo contains fire, smoke, or a person. In contrast, Regression deals with the estimation of continuous values, i.e., what the distance from the camera to the object is given the visual input.

The central part of the most deep learning models in computer vision is made by supervised learning. However, its profitability is mainly reliant on the label quality and information quantity for the data available. The key to the model’s performance under actual circumstances is the availability of extensive most comprehensive datasets, which are a mixture of all possible varieties and have well-done annotations.

2.1.2 Unsupervised Learning

Data that is unlabeled or has no predefined outcomes is used in unsupervised learning [1]. The model is provided with the data so that it can autonomously identify hidden patterns or structures. This kind of learning is particularly useful for data exploration, complexity reduction, or similar data points clustering.

Clustering, which is the grouping of similar items like customers with the same behaviour, and dimensionality reduction, which focuses on the most important features of complex datasets are the most common unsupervised tasks. Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) are the two main techniques used for this purpose, and they are often applied in various areas such as market analysis, anomaly detection, image processing, or gene studies in biology.

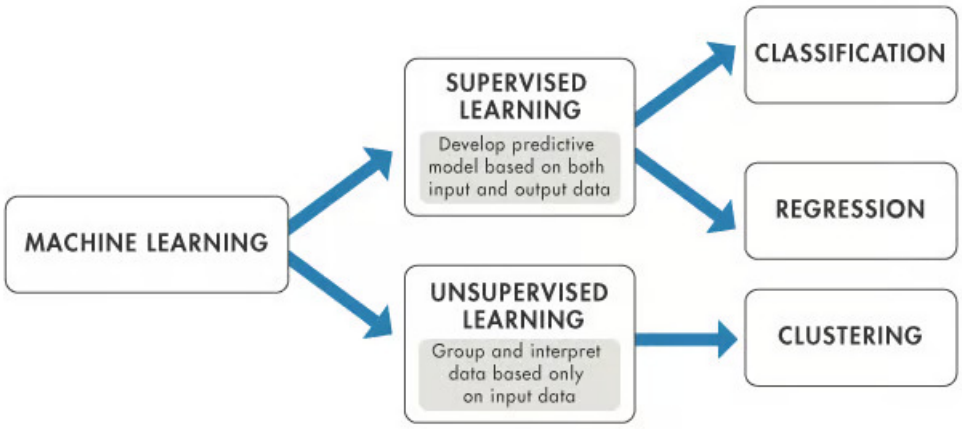


Figure 2.1: Comparing supervised and unsupervised learning [1]

2.2 Deep Learning

Deep learning [2] is actually a subfield of machine learning which looks deeper into the data and is quite popular. Deep learning is the technical field of artificial intelligence that strongly emphasizes the learning of data status using the method of a hierarchical structure. Deep learning's philosophy is rooted in the brain's architecture, that makes it easy to carry out most of the natural learning processes. The somewhat long chains of simple computational nodes are provably effective at discovering features that may or may not be useful in the next stages of the system's input representation, without the need for manual feature engineering.

It's fair to say that artificial intelligence (AI) has transformed a lot in the last ten years, mostly thanks to the process of deep learning. It's been the most instrumental AI technology that has practically solved old AI problems completely or at the least to a great extent, for example problems like semantic parsing, transfer learning, natural language processing, and computer vision. Three main reasons are behind deep learning's going mainstream:

- The massive improvements in processing power, especially through the use of Graphics

Processing Units (GPUs),

- The decreasing cost and increased accessibility of high-performance computing hardware, and
- Continuous innovation in machine learning algorithms that make training deep networks more efficient and scalable.

In the field of computer vision, we often use Convolutional Neural Networks (CNNs), which are capable of efficiently processing a large amount of image data, and they are a well-liked type of Artificial Neural Network (ANN).

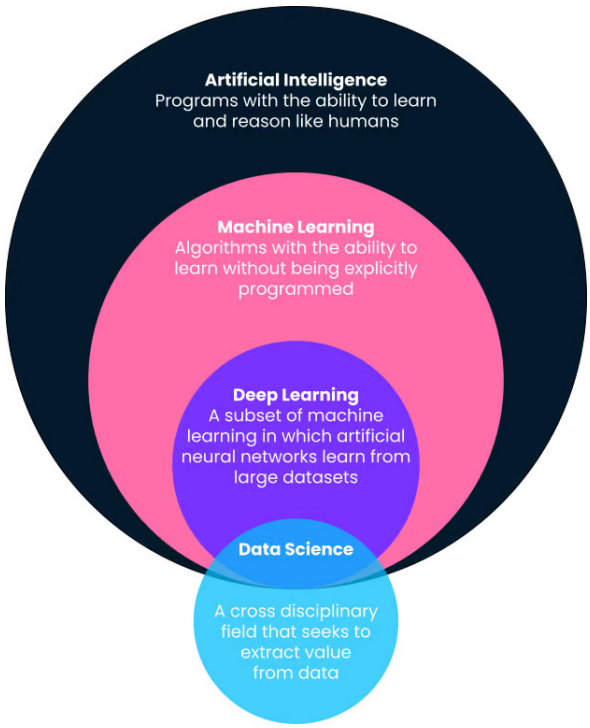


Figure 2.2: Comparing different terms [1]

2.2.1 Artificial Neural Networks

ANNs [3], commonly called neural networks, are computing structures that mimic the workings and functioning of the human brain. The layers of interconnected nodes that are the neurons, process the data through a network of input signals, weights, and biases. They amount to the system's "silicon neurons", which act together in the recognition, classification, and description of data patterns.

In most cases a neural network is comprised of the three layers that are input, hidden, and the output layers. The input layer is where data are first introduced to the network, and on the other hand, the output layer is the final prediction. The layers that are in between, i.e. hidden layers, have neurons that gradually change and enhance the data as it passes through the net. A neural network with many such hidden layers is called a deep neural network.

The method of passing data from the input to the output of the network is called *forward propagation*. The process of each neuron is such that the inputs are multiplied by weights, summed with a bias term, and then passed through an activation function to bring in the non-linearity. The result is subsequently sent to the next layer.

Another procedure is employed to fortify the model's performance, it is called *backpropagation*. This algorithm measures the difference between the output that is predicted and the one which is the ground truth, then it passes this error numerical through the layers but backward this time. Using optimization methods such as gradient descent, the model adjusts its weights and biases layer by layer to minimize the prediction error. Through repeated cycles of forward and backward propagation, the network gradually learns to make more accurate predictions.

The computation that lies behind the training process of deep neural networks is extensive. It has the need for a big processing power because of the huge number of parameters and complicated operations that are performed. GPUs that provide high performance are the most widely used hardware for such tasks as they can execute parallel computations on thousands of cores with high memory capacity. Furthermore, cloud-based distributed computing solutions are capable of delivering scalable resources for deep learning applications.

Moreover, the majority of the cutting-edge deep learning applications today are implemented using the specialized frameworks like TensorFlow and PyTorch. These frameworks are built to offer convenient and more performant abstractions for the process of building, training and deploying neural networks at any scale.

2.2.2 Convolutional Neural Networks (CNNs)

CNNs [3], also known as ConvNets, are a particular type of artificial neural networks primarily intended for the tasks of computer vision. Their structure is ideal for dealing with visual data such as pictures and videos, as they can automatically discover local patterns and hierarchical representations. CNNs have been a driving force behind the huge progress

in image classification not to mention object detection, facial recognition, and other pattern recognition applications.

In the typical CNNs, we have numerous layers which usually start with an input layer, relay on several hidden ones, and finish with an output layer. Each layer comprises of nodes (or neurons), and to make the point clear, every node is accompanied by a set of weights as well as a threshold. As long as the weighted input to a node goes beyond this threshold, the node gets activated passing the information to the next layer. The less composed signal is completely suppressed. This mechanism allows the network to learn non-linear decision boundaries and gradually refine its predictions.

What distinguishes CNNs from traditional neural networks is their ability to exploit the spatial structure of image data through three key types of layers: convolutional layers, pooling layers, and fully connected (FC) layers.

- *Convolutional layers* apply a set of learnable filters to the input data, scanning across local regions and extracting low-level features such as edges, textures, or simple shapes. These filters are optimized during training to capture the most relevant patterns.
- *Pooling layers* are used to reduce the spatial dimensions of the feature maps generated by convolutional layers. This downsampling step helps to control overfitting, reduce computational load, and retain the most essential features.
- *Fully connected layers* appear toward the end of the network and operate similarly to traditional ANNs. They integrate all extracted features to make the final classification or prediction.

Different layers of a CNN learn from the incoming data and in the process, the CNN extracts increasingly abstract and complicated features. Typical early layers are used to find out the key elements such as edges or color gradients, while the deepest layers signify the patterns like object parts or entire shapes. This multi-layer structure allows CNNs to build a durable understanding of the visual content.

Earlier than the introduction of CNNs, feature extraction was mainly executed manually in the computer vision environment, requiring domain-specific techniques and heuristics. CNNs have eliminated this by using the end-to-end learning approach, where both the feature extraction and classification are learned from the data at the same time. This not only makes the classifier more accurate but also much more scalable and adaptable to different tasks.

However, despite their strengths and qualities, CNNs still come with downsides. They are computationally intensive and quite often, extensive resources, e.g., many GPUs, are needed for training. Furthermore, it is a challenging task to design and tune a CNN architecture, which requires the user to opt for proper hyperparameters and often necessitates expertise in both machine learning and domain-specific knowledge. Still, the benefits like fewer parameters, feature recognition with higher efficiency, and better results on high-dimensional data give CNNs a lead position over the traditional methods in present computer vision research and applications.

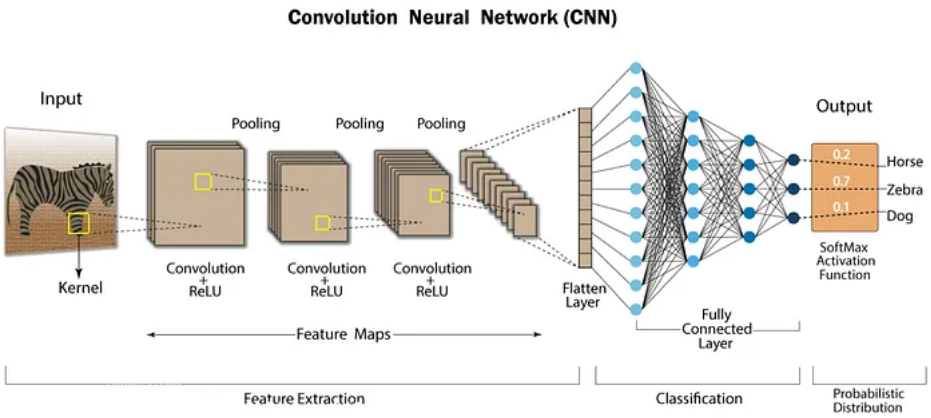


Figure 2.3: Convolutional neural network (CNN) architecture [4]

2.3 Computer Vision

Computer vision [3] is a branch of AI that deals with the subject of self-improvement in the present time. It is about giving computers the ability to interpret and understand visual data such as the images, video streams, or other sensor-based inputs. This involves a lot of tasks such as image classification, object detection, semantic segmentation and many more. While artificial intelligence gives to computers the ability to “think”, computer vision gives them the ability to “see”, observe and understand their environment.

At the root, computer vision is using the idea of machine learning and neural network models, mainly CNNs, to pull out the important patterns from visual content. These models are being educated to input the visual data, process through the data, recognize useful features, and then take appropriate actions or activate certain functions according to the detection. For instance, a computer system could easily detect and signal any irregularities, deviations or insecure situations that may occur in real-time due to which it becomes especially helpful in

monitoring, checking the quality of products, and coordinating safety-related aspects.

One of the major benefits of computer vision systems is the speed and accuracy they employ during the visual inspections. Once they have been trained, such a type of systems can then check and classify the images or the production assets which are thousands in number every minute, thus, they can spot tiny defects or detect the gentle changes that may not be noticed by the human eye. This situation then identifies this technology as very necessary for use in a lot of diverse industrial sectors such as the manufacturing, automotive, energy, utilities, health, and robotics industries.

The performance of a computer vision model is closely tied to the dataset□s volume and accuracy that is utilized for training . To provide the computer vision model with the ability to identify particular objects, for example, fires, a huge and varied dataset comprising of labeled images of fires must be trained including both normal and images showing different fire situations (e.g., small or large fires).

This is how the model can have sufficient exposure to the class distribution and learn the necessary features that distinguish the different classes and that allow the building of a robust internal representation of the visual categories.

With traditional programming, a developer is the one who sets the rules for recognizing objects in computer vision, whereas the latter is based on algorithmic learning. Through the repetitive scanning of their training data, models progressively get better at recognizing patterns and correlations. This type of learning is also known as self-supervised or data-driven. It is the system's capability of generalizing outside the training set and providing reliable predictions for new, unknown data.

The development of computer vision deriving from the availability of more complex models, greater amounts of data, and stronger computational resources continued. Combined with deep learning, it has become one of the most impactful technologies for enabling intelligent perception in real-world systems.

2.3.1 Object Detection

Object detection [4] is an essential computer vision technology that utilizes deep-learning models for the purpose of not only object identification but also location finding in the digital content, e.g., images or videos. If given an input image, a model for object detection comes up with the bounding boxes, which are the rectangular regions, encompassing each object

detected, and along with the boxes, the model also gives labels describing the content of each object.

An image may include several objects, all separately positioned in different locations, like several cars spread all over the street, or numerous buildings in the city layout. This idea of performing object detection on multiple items in a single image is referred to as multi-object detection. The function of the model is to recognize not only the class of each object but also its position on the image in a very detailed manner.

A bounding box is a square figure formed around the object in the image which indicates the position of the object in the image. Usually, it is represented by four features such as the box coordinates (center point, or the top-left corner that could vary with a model and define the width and height of the object). Deep learning models refine these bounding boxes through a process called regression, where the network predicts adjustments to the box's position and size to better fit the object.

For example, in wildfire monitoring applications, object detection can be used to locate and distinguish between fire and smoke within aerial images. The model might draw bounding boxes around patches of fire and smoke separately, labeling them accordingly, which helps responders assess the situation more effectively.

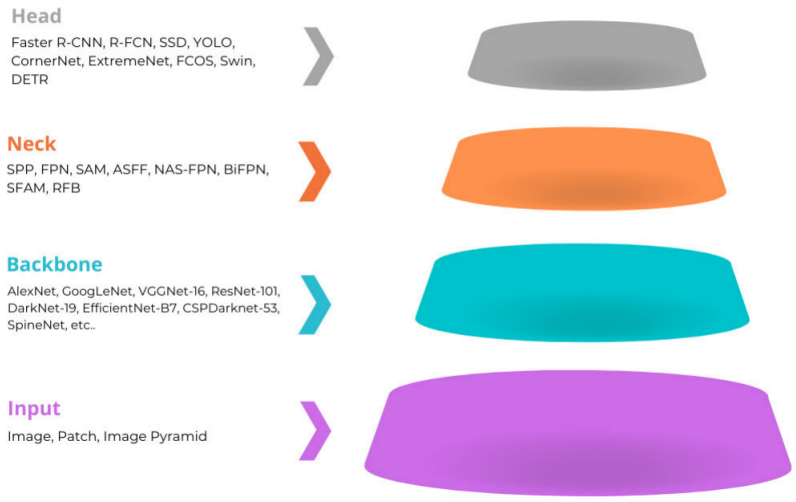


Figure 2.4: Object detection components [5]

As shown in Figure 2.4 an object detection system typically consists of several key components, each responsible for a specific part of the detection pipeline. These components are outlined below:

- *Input Image:* The system begins with an input image or video frame, which may come from RGB, thermal, grayscale, or multi-channel sources.
- *Backbone (Feature Extractor):* A deep convolutional neural network (CNN) is the foundation of the process that extracts both high-level and low-level features from the original image. The most commonly used backbone architectures include ResNet, Darknet, EfficientNet, and VGGNet. The output is the set of feature maps that represent textures, edges, and semantic structures in the image.
- *Neck (Feature Aggregation):* This part is responsible for the combining process of feature maps at various scales, thus making the detection of smaller and larger objects better. The prominent neck architectures are Feature Pyramid Network (FPN), Path Aggregation Network (PANet), and Bidirectional Feature Pyramid Network (BiFPN).
- *Head (Detection Module):* The head processes the aggregated features to generate bounding box predictions, class scores, and object confidence values. There are two main types of detection heads:
  - *One-stage detectors* (e.g., YOLO, SSD, EfficientDet) make predictions directly from feature maps.
  - *Two-stage detectors* (e.g., Faster R-CNN) in the first stage, the detection process is broken down into two tasks, in which a Region Proposal Network (RPN) identifies candidate regions of interest (ROIs) that have a high likelihood of containing objects while in the second stage a separate network operates on these ROIs to perform correct classification and bounding box regression.

Algorithm 1 Non-Maximum Suppression (NMS)

Procedure *NMS* (list of predictions P, IoU threshold θ)
Initialize empty list *keep*
while P is not empty **do**
    Select the prediction p in P with the highest confidence
    Remove p from P and add it to *keep*
    for each remaining prediction r in P **do**
        if IoU(p, r) $> \theta$ **then**
            Remove r from P

```
        end if
    end for
end while
Return keep
End Procedure
```

Before finalizing predictions, each object detection model applies the Non-Maximum Suppression (NMS) [6] algorithm within its *head* to eliminate redundant bounding boxes and retain only the most confident detections. The process operates as shown in Algorithm 1.

In simpler terms, this algorithm works by picking the prediction with the highest confidence score and eliminating any overlapping predictions that exceed the IoU threshold. The name ”non-maximum suppression” comes from the fact that it retains the maximum confidence prediction and suppresses others with significant overlap.

2.3.2 Object Tracking

Object tracking [7] is an important computer vision technique where a program detects objects and follows their movements through space or across different camera views. It is capable of identifying and tracking multiple objects within a scene. For example, in a football broadcast, object tracking can continuously monitor the position of the ball as it moves across the field.

This technology is widely used in augmented reality and other real-time applications to estimate or predict the location and relevant attributes of moving objects.

Although related, object detection and object tracking serve different purposes. Object detection tries to locate and then label objects mostly on images or frames extracted from a video. Normally, these algorithms draw the bounding boxes around the detected areas. On the other hand, object tracking algorithms guarantee the continuity of the same objects from one frame to the next in the video and, therefore, maintain the identities of these objects while also showing us their paths that have been taken thus far. The typical process of object tracking involves several key steps:

- *Input:* Receiving video input from a file or live camera feed, with preprocessing applied to each frame for consistency.

- *Object Detection:* Applying detection algorithms to identify objects and draw bounding boxes around them.
- *Labeling:* Assigning unique IDs to each detected object to distinguish them.
- *Tracking:* Continuously updating the position and movement of each object through the video frames.

Object tracking comes under the two main categories, namely image-based and video-based tracking. The first one, i.e., Image-based tracking, is about locating and anchoring whole 2D images or objects in a scene. Typically, it is used in augmented reality to place virtual objects in the real world, for instance, enabling consumers to see furniture in their homes through e-commerce platforms. This method has some room for error in the spatial sense, up to a few centimeters. Video-based tracking, also known as real-time tracking, involves predicting the position of objects frame-by-frame using both spatial and temporal information. It is further divided into:

- *Single Object Tracking (SOT):* Focuses on following one object throughout a video, starting from a defined bounding box in the initial frame.
- *Multiple Object Tracking (MOT):* Involves detecting, labeling, and tracking several objects simultaneously across video frames, maintaining their unique identities until they leave the scene.

Figure 2.5: Object detection vs Object tracking [8]

2.3.3 Data Augmentation

Data augmentation involves generating new versions of images through various transformations like rotation, zooming, flipping, and adjustments to brightness, contrast, and more.

This technique aims to expand and diversify the training dataset, thereby enhancing the accuracy and robustness of the object detection model.

2.3.4 Image Registration

Image registration is the process of aligning multiple images of the same scene into a common reference frame. It plays a crucial role in systems that utilize data from different sensors, such as RGB and thermal cameras. Aligning such images enables accurate comparison and combination of information captured in different modalities.

In multi-modal setups like the one addressed in this thesis, traditional feature-based registration methods may not always be effective due to the differences in appearance between RGB and thermal imagery. Instead, registration can be guided by corresponding reference points observed in both image types. This alignment ensures that subsequent processing stages, such as object detection and data fusion, operate on geometrically consistent input.

2.4 UAV

An Unmanned Aerial Vehicle (UAV) [9], also known as a Remotely Piloted Aircraft (RPA) or drone, is an aircraft that operates without a human pilot onboard. It can be maneuvered remotely, fly autonomously via onboard computers, or use a mix of both. UAVs are one of the components of the broader Unmanned Aerial System (UAS), which also consists of the aircraft itself and the connected ground equipment and software enabling the operation without the direct intervention of humans.

The order is mostly followed in an existing area through the Ground Control Station (GCS) and the Remote Controller (RC). The GCS serves as a command center, aids in flight planning, monitoring, and real-time communication between the UAV and its environment. It usually comes with interfaces for mission planning, live video feeds, telemetry data, and communication links with the UAV. The RC is a handheld gadget used by an operator for the transfer of commands and the receipt of data from the drone.

A unique subset of UAVs can be defined as those with some level of capability to be able to perform missions without intervention from humans. To complete the mission, the UAV must be equipped with *perception*, the ability to pick up and process data from its environment via onboard sensors like a camera, lasers, and environmental detection sensors. That is to say,

the UAV is using its senses or feelings to study the environment and accordingly make the required decisions.

A *mission* denotes the specific task that the UAV is given to do, which could range from mapping and exploration to life-saving operations, traffic monitoring, firefighting, and many other functions. It is common to see a group of UAVs working in concert to perform complex missions. The UAVs often form multi-UAV systems that execute the mission as a whole. These are often used in reconnaissance, monitoring, exploration, and surveillance operations. A team comprising multiple UAVs is preferred to a single one since a group can execute missions more rapidly and efficiently. Such teams are less vulnerable to detection, and if a single UAV fails, it can be quickly replaced by another one in the team.

The UAVs execute their missions with high accuracy by letting automatic pilots follow a sequence of *waypoints*, which are the predefined intermediate positions that the vehicle takes towards its final destination. Each waypoint includes a tolerance threshold, allowing the UAV some flexibility in its path while ensuring mission objectives are met.

Perception, mission planning, and waypoint navigation are the three main parts that give the UAVs the ability to work independently in different environments efficiently.

2.5 Sensor Technologies

Drones utilize sensors to perform one of their most critical functions – to detect and measure various environmental inputs such as light, heat, motion, humidity, and pressure. The sensors then transform the physical phenomena into electrical signals which the UAV's systems interpret to know the environment in which they are present. The application of this gathered data to UAVs enables them to move, fly the right path, and accomplish required mission tasks safely and effectively. The sensors that are used for this thesis are RGB camera, infrared/thermal camera and Global Positioning System (GPS).

2.5.1 RGB Camera

RGB cameras capture images through a color filter array (CFA) that is used to realize visible light mostly by the Bayer BGGR design [10]. The array is set on the sensor's pixels with each pixel designed to respond to one of the primary colors: red, green, or blue.

The color sensor, the nucleus of an RGB camera, is typically a Charge-Coupled Device

(CCD) or a Complementary Metal-Oxide-Semiconductor (CMOS). It is in charge of capturing the light coming from a scene and converting it into an electronic signal. The sensors take care of the energy within the visible wavelength spectrum falling roughly between 400 and 700 nanometers, enabling image capture under natural lighting conditions.

When it comes to the distribution of individual pixel colors, it is important to note that it is the structure of human vision that mainly enforces the usage of green pixels in the design of an image sensor. This is because the human eye has a higher sensitivity to the green wavelength than to red or blue ones and, as a result, more green pixels will significantly help in making the image look real. Demosaicing is a sophisticated technique a camera employs to produce high-quality image of natural color, i.e., the technique where the data of individual color pixels is combined to reconstruct the details and colors of the scene virtually identical to the way the human eyes see it.

2.5.2 Infrared Camera

All objects [11] give out electromagnetic radiation, most of which are invisible to the human eyes. Our visual system can only see the colors within a thin band called the visible light spectrum. Yet, thermal imaging drastically widens this range by enabling us to observe and record thermal energy that was hitherto invisible. This form of radiation is called infrared radiation, as shown in Figure 2.6.

It can be said that the infrared radiation of an object positively correlates with its temperature. Infrared thermal imaging cameras facilitate the measurement of temperature from a distance, eliminating the need for direct contact with potentially hazardous materials. Since many machines and electronic devices emit heat before failing, infrared cameras play a key role in preventive diagnostics and frugal maintenance.

One advantage of such cameras is that they require no visible light. Every object with a temperature above absolute zero releases infrared radiation regardless of its color as can be seen in ice cubes, too. This means infrared cameras can reliably capture thermal images whether in darkness or bright light.

Infrared cameras are non-contact devices that detect thermal energy and convert it into electrical signals. These signals are then processed to generate images representing temperature variations. Beyond simple visualization, these systems enable precise temperature measurements, allowing users to assess not just the presence of heat but also its relative intensity.

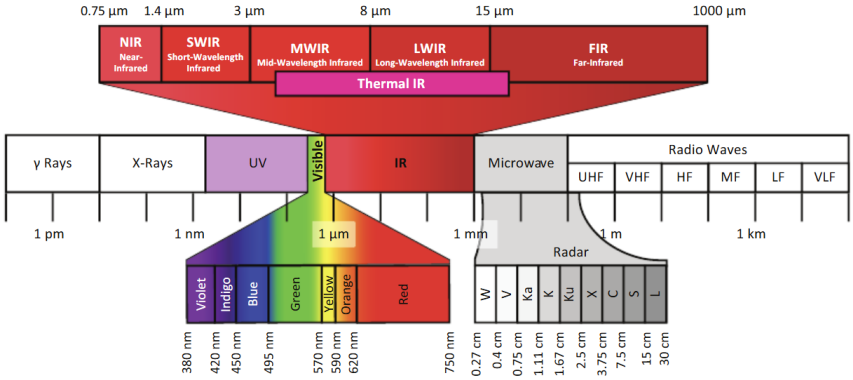


Figure 2.6: Depiction of the electromagnetic spectrum [12]

As shown in Figure 2.6, the infrared (IR) spectrum is divide into several categories, each with distinct characteristics and uses:

- *Near-Infrared (NIR):* Operates in the 750–1400 nm range [13]. Mainly used for vegetation and soil monitoring, as healthy plants strongly reflect NIR light. Also helps differentiate water from land in environmental imaging.

- *Short-Wave Infrared (SWIR):* Covers 1.4–3 µm [14]. Captures reflected infrared light, useful for imaging through haze or smoke. Enables material inspection, moisture analysis, and mineral detection. SWIR cameras often use uncooled InGaAs sensors.

- *Mid-Wave Infrared (MWIR):* Ranges from 3–8 µm [14]. Detects thermal emissions from hot objects like engines or industrial equipment. Requires cooled detectors (e.g., InSb), offering high thermal sensitivity for high-temperature applications.

- *Long-Wave Infrared (LWIR):* Operates in the 8–15 µm range [14]. Detects heat from ambient objects (humans, buildings) using uncooled microbolometers (e.g., vanadium oxide). Widely used in thermal imaging for surveillance and firefighting.

- *Far-Infrared (FIR):* Extends from 15 µm to 1 mm [15]. Represents low-energy thermal radiation from objects like the sun or human body. Used in deep-tissue medical therapy, thermal energy transmission, and infrared astronomy.

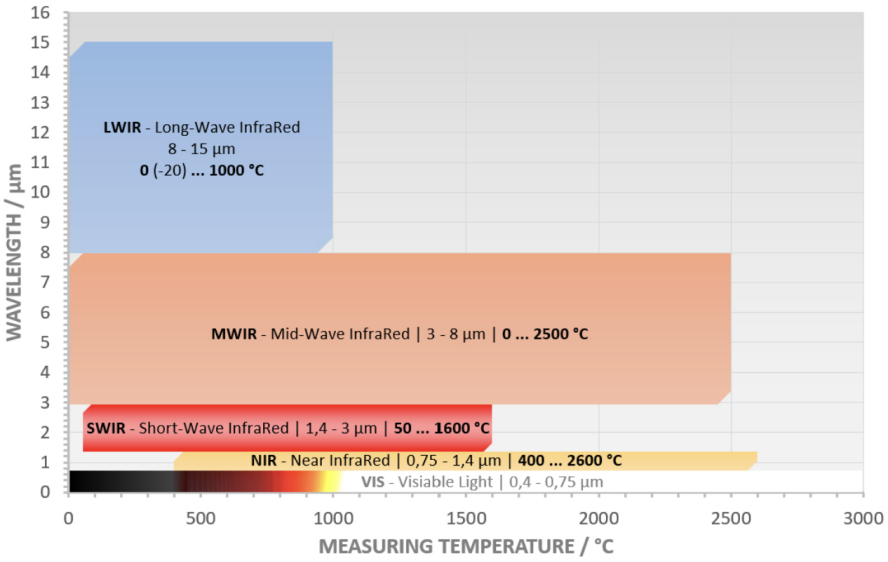


Figure 2.7: Infrared wavelength bands and their corresponding temperature ranges [16]

As shown in Figure 2.7, a wide variety of temperatures are presented on the infrared spectrum, each of them representing a specific range of wavelengths. If the temperature ranges and properties that were just represented are regarded, LWIR cameras are thought to be the most optimal devices for the execution of the UAVs missions. Data obtained by the description of the LWIR sensors work has indicated they work together with the thermal radiation that is generated by the objects at normal outdoor temperatures such as the human body, animals, and fire. In addition to this, uncooled LWIR cameras are usually the ones used in this situation, which is cost-effective and energy-saving and will reduce functionality and thus system complexity is downsized and the factors that are crucial for air platforms like UAVs. On the contrary, the temperature ranges of NIR and SWIR are rather the ones that are not suitable for thermal radiation detection capabilities due to the fact that they represent the temperatures at which objects begin to emit NIR or SWIR radiation, following the blackbody emission principles. This thesis is focused on the selection of LWIR thermal cameras to perform airborne thermal imaging and monitor tasks, which satisfy three criteria of sensitivity, practicality and operational efficiency.

2.5.3 GPS

The Global Positioning System (GPS) [17] is a space-borne system that supports definite navigation, precise location, and time (PNT) measurements globally. It is carried by satellites

encircling the earth and sending signals to GPS receivers. These signals enable users to know exactly where they are and what time it is at any location on the globe. GPS operates as follows: it receives signals from the satellites which are then used to determine the exact position and time of the receiver. GPS finds application in various fields such as transportation like vehicles, airplanes, and ships, as well as in agriculture, surveying, and mapping. The system comprises three main segments: the space segment, which contains the satellites; the control segment, which oversees the entire system; and the user segment, which involves the people and the devices that are using GPS for PNT measurements.

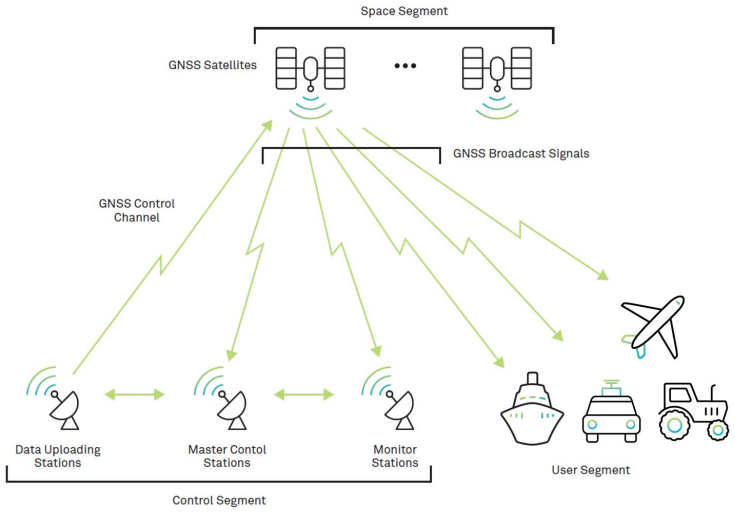


Figure 2.8: GPS structure [17]

2.6 Simulation Environment

2.6.1 Unreal Engine Overview

Unreal Engine (UE) is a high-quality platform that individuals working in various sectors, including gaming and robotics simulation, rely on for their real-time 3D creation needs. It has integrated tools for high-quality graphic rendering, simulation of physical phenomena and intelligent systems, and is entirely customizable, thus it can be used to develop real-world environments for UAV simulations in a photo-realistic way. Unreal Engine is able to develop photorealistic 3D environments with precise textures, lighting, and effects that are necessary for robotics and autonomous systems to be tested accurately.

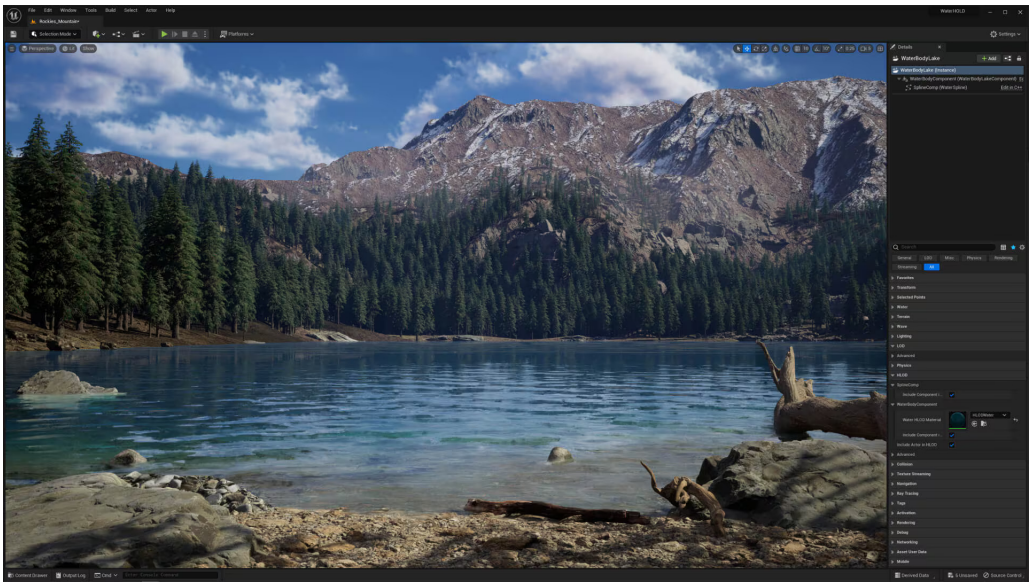


Figure 2.9: Unreal Engine 5 editor [18]

2.6.2 AirSim Plugin

AirSim, originally developed by Microsoft, is a simulator designed specifically for testing autonomous vehicles, including drones. AirSim runs on the Unreal Engine platform and presents a realistic situation for a drone to be controlled and tested. The virtual environment eliminated the need for physical hardware, and that was the main benefit of the Unreal Engine-based AirSim that was used during this project. AirSim realizes the vision of enabling end-to-end testing, especially navigation and control algorithms and integrates them with physical sensors through the on-board computer and, thus, enabling different kinds of experiments on UAVs. Additionally, the thermal cameras used in AirSim are custom-built from RGB cameras, which will be analyzed further in the implementation section. While development of AirSim was officially stopped by Microsoft, the project has since been continued by the community, with individuals and organizations contributing to its ongoing development. This thesis uses the `Cosys-AirSim` [19] version, which represents the work of `Cosys-Lab`. This fork of AirSim not only fully compatible with newer Unreal Engine versions but also developed with more advanced features for UAV simulation in indoor/outdoor scenarios. It is worth noting that AirSim is a part of Unreal Engine as a plugin, with Unreal Engine being in charge of the simulation environment, and AirSim being the controller of the quadrotor, allowing multiple drone systems to work together efficiently in testing and simulation.

Figure 2.10: AirSim simulator [20]

2.7 ROS 2 Framework

2.7.1 Overview of ROS 2

ROS 2 [21] is an open-source robot framework that overcomes the success of its predecessor, ROS 1. It is a set of software libraries, tools, and communication frameworks for the easy implementation of robotic applications. What ROS 1 did was to introduce robotics to a new era with its modular design and a long list of sensors, actuators, and algorithms that it could use efficiently. Nevertheless, ROS 2 has taken the challenges of ROS 1 and brought the matter to a different level by introducing notable enhancements to it.

	ROS 1	ROS 2
Middleware	TCPROS	DDS (Data Distribution Service)
Network Communication	Less robust	Quality of Service (QoS), Multicast
Real-time Support	Not officially supported	Full support with RTOS
Security	Basic or nonexistent	DDS Security
Software Compatibility	Linux only	Linux, Windows, Mac, RTOS
Scalability	Limited	High, ideal for fleets

Figure 2.11: Comparison of ROS 1 and ROS 2 [21]

As shown in Figure 2.11, ROS 2 introduces several significant improvements over ROS 1, addressing many of its limitations:

- *Middleware:* ROS 1 uses TCPROS, a custom protocol based on TCP, whereas ROS 2 adopts DDS (Data Distribution Service), a widely-used standard that enables more robust and scalable communication.
- *Network Communication:* ROS 2 supports advanced communication features such as Quality of Service (QoS) and multicast, making it more suitable for complex and distributed robotic systems. In contrast, ROS 1's communication is less robust and lacks such features.
- *Real-time Support:* ROS 1 does not officially support real-time operation. ROS 2, however, offers full support with real-time operating systems (RTOS), making it more appropriate for time-sensitive robotic applications.
- *Security:* ROS 1 provides minimal to no built-in security features, while ROS 2 includes DDS Security, enabling authentication, encryption, and access control.
- *Software Compatibility:* ROS 1 is limited to Linux platforms. In contrast, ROS 2 supports multiple platforms, including Linux, Windows, macOS, and RTOS, increasing its accessibility and flexibility.
- *Scalability:* ROS 2 is designed to scale efficiently and is ideal for large deployments such as fleets of robots. ROS 1, on the other hand, is limited in this regard.

2.7.2 Relevance in Robotics and Simulation

ROS 2 is currently a highly preferred choice in the field of robotics for physical as well as simulation models because of its stable communication system, flexibility, and potential to be scalable. It can be employed in robotics to supervise and operate various parts of self-governing systems such as aerial vehicles and industrial robots. ROS 2 facilitates communication between sensors, actuators, control algorithms, and navigation systems. The interfacing of multiple subsystems is the main advantage of the ROS 2 platform. Hence, a robot running on ROS 2 is a formidable force when it comes to the accomplishment of difficult tasks with the required level of fault tolerance and speed.

ROS 2 is the major platform for simulation, it connects the digital and real worlds of robotics via different approaches (like in simulation platforms such as Gazebo and Unreal Engine). ROS 2 not only allows engineers to test out their new ideas, confirm their designs but also provides an opportunity to experience real-life occurrences without any practice of the material threats or high expenses. The real-time communication function within ROS 2 is the source of this ultra-precise and exact robot and the virtual world's synchronization, and therefore, the best match for the realization of robot systems processes (e.g., smart navigation, object recognition, etc.) in a real, practical scenario. ROS 2's plus point is the fact that it runs on multiple devices, and the tools and libraries are consistent across different hardware and simulation platforms, thus it becomes an integral part of robotics R&D. The achievement of robotics development and research is further guaranteed by the compatibility of ROS 2 across different platforms, where developers are able to use the same tools and libraries from a wide range of hardware and simulation platforms.

Chapter 3

Related Work

The paper [22] reports a thorough investigation into UAV-based wildfire detection using real fire captured RGB and IR image data. It was also beneficial to this thesis that the article outlined the major problems related to satellite and lookout-based fire monitoring systems, which could be the conventional systems' larger spatial resolution errors, lower frequency of measurements, or for example, hidden fires due to smoke and plants. On the other hand, UAVs not only were capable of performing the tasks but also the high resolution and the possibility of getting the data in real-time demonstrated it is the right choice for the current work. A critical contribution of the paper was its use of co-registered RGB and IR (thermal) imagery, annotated for fire and smoke detection, which directly aligns with the multi-modal approach employed in this thesis, except for the missing human and animal annotations. The authors mention that the data collected via the fusion of sensors help in a consistent way a detection system to be developed in various environmental situations. These became the basis for the design of the data fusion module in this thesis, combining RGB and thermal predictions for more accurate detection.

The difference between our working method and the one presented in this article is the fusion technique. As illustrated in Figure 3.1, the paper implements both early and late fusion techniques directly within the model architecture. n the early fusion way, RGB and thermal images are first combined at the input level to result in a six-channel input (or four-channel if the thermal image is grayscale). In the late fusion way, feature maps from two separate processing streams, one for each modality, are combined after being processed independently and the final classification is done through a fully connected layer that integrates both modalities. However, such an approach requires perfectly aligned RGB and thermal im-

age pairs across all target classes, including fire, smoke, humans, and animals. However, the FLAME2 [23] dataset, which is used in the paper, does not present all the data completely and in line with our case, especially for humans and animals, which are a part of our application.

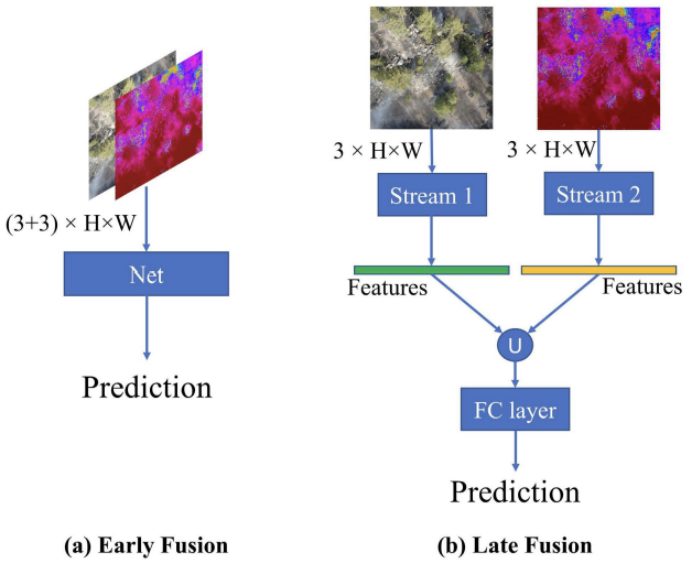


Figure 3.1: Fusion techniques [22]

However, the absence of a completely aligned dataset containing the necessary classes led us to employ a different approach which we found to be the preferable option. We designed and trained two distinct models: one for the RGB images and the other for the thermal images. After that, we used a special late fusion method at the stage of prediction which is done by bringing together the results from the models belonging to different modalities only after they have operated. The modular character of this solution preserves the alignment of the data where it exists, and at the same time, it permits each model to be trained independently on datasets related to different modalities, which may contain different environmental scenery and class distributions. As a result, the models can be more generalized and robust, rather than overfitting to a single, limited dataset with low environmental diversity.

Furthermore, one of the key objectives of this thesis was to compare different object detection models based on their accuracy and real-time performance in UAV-based wildfire scenarios. Developing modified models that can receive both modalities at the same time, for example, by using early or embedded late fusion, would have been very complex and would have deviated the attention from the comparative evaluation. Further, no time was available for the task of making such hybrid models work and be fair when comparing different archi-

tectures. Therefore, the current modular fusion design was chosen for its clarity, flexibility, and alignment with the thesis objectives. It is, however, a fact that integrating both modalities directly to the main model can be seen as a future improvement of the system that is also a natural optimization of the current system and hence, it provides a more unified approach to the current implementation.

The paper [24] contributed significantly to this thesis by providing a multimodal dataset that was used during the training of the object detection models. Although the dataset does not include pre-aligned RGB and thermal image pairs, the paper presents a detailed method for image registration, which helped guide the alignment process implemented in this work. The dataset provided was particularly useful for the detection of fire and smoke in both RGB and thermal modalities, and it was even more so with the provision of numerous examples of visual information in real-life conditions. Further evidence that researchers provided was the configuration of the camera and calibration that allowed us to also understand all the sensor characteristics and alignment requirements. Nevertheless, the dataset is devoid of annotations for classes such as humans and animals, which are very important for this project making the research question combining this data in this paper difficult to carry out. Trying to fill in this gap, information other than the main sources was introduced to enlarge the training set and thus reinforce the system's ability to recognize all object classes that are important.

In these papers [22], [24], [25], [26], [27], [28], the authors extensively evaluated various deep learning-based object detection models, including YOLOv3, YOLOv5, YOLOv8, SSD, Faster R-CNN, ResNet as backbone for classification and EfficientDet D1, within the context of UAV-based fire and smoke detection. Their work provided valuable insights into which models balance detection accuracy and real-time inference performance, a critical requirement for deployment in UAV systems. While some of the comparisons were conducted solely on RGB imagery, others included thermal data, offering a broader view of model capabilities in different sensing conditions. The results helped highlight each model's strengths and limitations in dynamic aerial environments. These studies guided the selection of specific models to be evaluated and compared in this thesis and informed the overall evaluation methodology. They also emphasized the importance of using appropriate input image resolutions adapted to the limitations of the UAV for a real-time application. Although the datasets used in these papers did not contain all target classes (e.g., humans and animals), the experiments offered concrete benchmarks for developing detection pipelines tailored for real-world

UAV monitoring applications.

Chapter 4

ML-Based Wildfire Detection

4.1 Introduction

The detection of wildfires using machine learning models deployed on UAVs is a challenging task that requires not only robust datasets but also efficient, accurate, and hardware-aware model design. The chapter introduces the whole pipeline of object detection for fire, smoke, human, and animal detection in the context of the two imaging modalities: RGB and thermal. The process begins with the assembly and preprocessing of the dataset of multiple sources and then goes on to training and evaluating a variety of deep learning-based object detection models. The evaluation part comprises of a thorough performance comparison in accuracy, inference time, resource utilization, and robustness.

With all models evaluated, the best detector for each of the two datasets, RGB and thermal, is identified. The F1-score analysis performed on both validation and test datasets allows determining the confidence level at which the selected model will operate, which ensures that it delivers real-time and balanced predictions. Eventually, the selected models are processed with the aid of NVIDIA TensorRT to be capable of running on edge devices. On comparing FP16 and INT8 with the original model in FP32, the trade-offs between detection accuracy and performance are made. Using the benchmarking, the model is first optimized for a high-performance (NVIDIA RTX 4090) platform, and secondly for an embedded (Jetson Orin Nano) platform to prove real-time feasibility from different hardware aspects of device.

4.2 Datasets Construction and Preprocessing

4.2.1 Introduction

Detection system based on deep learning is that the major correctness of the data, the variety and the accuracy of annotations have a significant impact on the design of training data. In this thesis, in order to form a representative dataset, the data was combined and preprocessed from a multitude of freely available resources. The dataset created was of two main types, namely RGB images and thermal (infrared) images of objects, which represented different objectives of the whole wildfire monitoring pipeline.

Just because the models used in this research were so diverse, e.g., YOLOv8s, Faster R-CNN with MobileNetV3, SSD with VGG16 and EfficientDet D1, it was a need to process the datasets in such a way that they may be compatible with three standard annotations: YOLO, PASCAL VOC, and COCO. This preprocessing step ensured compatibility with each model's training pipeline and allowed for direct comparison of detection performance across modalities and architectures.

The following subsections present the data sources and data preparation processes for each category in full detail, showcasing the essential measures that were adopted to build a useful and flexible dataset that would serve for wildfire detection and monitoring applications in the real-world.

4.2.2 Background

4.2.2.1 Annotation Formats

The RGB and thermal datasets used in this thesis were converted into three standardized annotation formats to ensure compatibility with different object detection models: YOLO, PASCAL VOC, and COCO. This subsection briefly describes the structure and key components of the YOLO, PASCAL VOC and COCO formats.

1. YOLO Format

In the YOLO labeling format [29], each image is associated with a `.txt` file stored in a `labels` directory. Each line of the file corresponds to one object and includes:

- *Class ID:* Specifies the class of the object.

- *Normalized Coordinates:* Contains the object's center coordinates (x_{center}, y_{center}), width, and height, all normalized by the image's width and height.

This normalization ensures that the annotations are independent of the image resolution. Each object in the image is recorded on a separate line, as illustrated below:

```
<object-class><x><y><width><height>
```

Figure 4.1: YOLO annotation record [29]

2. COCO Format

The COCO annotation format [30] is stored in JSON files, typically with separate files for training, validation, and testing datasets. It supports multiple annotation types such as:

- *Object Detection:* Bounding boxes and class labels
- *Keypoint Detection:* (not used in this thesis) for landmark-based tasks

For object detection, the relevant COCO fields include:

- *Images:* Metadata about each image, including filename, ID, and dimensions.

```
"images":[
         {"id":0, "license":1,"file_name":"001.png","height":960,"width":1280,
         {"id":1, "license":1,"file_name":"002.png","height":900,"width":1920,
         {"id":2, "license":1,"file_name":"003.png","height":960,"width":1280,
        ]
```

Figure 4.2: COCO annotation file part 1 [30]

- *Categories:* Class labels with numeric IDs and names.

```
"categories"= [
              {"id":0, "name":"Airplane", "supercategory":"none"},
              {"id":1, "name":"Airbus A220", "supercategory":"Airplane"},
              {"id":2, "name":"Boeing 314", "supercategory":"Airplane"},
            ]
```

Figure 4.3: COCO annotation file part 2 [30]

- *Annotations:* Bounding boxes, class IDs, segmentation (optional). The bounding box coordinates in COCO format follow the structure: `[x_min, y_min, width, height]`, where `x_min` and `y_min` define the top-left corner of the bounding box, and `width` and `height` represent its size in pixels.


```
"annotations": [
                {
                  "id": 0,
                  "image_id": 0,
                  "category_id": 0,
                  "bbox": [
                    555,
                    408,
                    201,
                    142
                  ],
                  "area": 28542,
                  "segmentation": [],
                  "iscrowd": 0
                },
                {
                  "id": 1,
                  "image_id": 0,
                  "category_id": 2,
                  "bbox": [
                    880,
                    468,
                    139,
                    134
                  ],
                  "area": 18626,
                  "segmentation": [],
                  "iscrowd": 0
                },
                {
                  "id": 2,
                  "image_id": 1,
                  "category_id": 1,
                  "bbox": [
                    551,
                    51,
                    180,
                    38160
                  ],
                  "area": 18626,
                  "segmentation": [],
                  "iscrowd": 0
                },
               ]
```


Figure 4.4: COCO annotation file part 3 [30]

3. PASCAL VOC Format

The PASCAL VOC annotation format [29] uses an XML file to store the annotations for each image. Each image has its own XML file, named identically to the image filename, and these XML files contain all bounding box details for the objects present in the image.

The annotations typically include:

- *Image Metadata:* Information such as the image filename, dimensions (width, height, depth), and the folder where the image is located.

- *Object Annotations:* For each object in the image, the following attributes are included:

- *Class Name:* Specifies the semantic category of the object (e.g., fire, smoke, human).
- *Bounding Box Coordinates:* Represented by pixel values of $x_{\min}, y_{\min}, x_{\max}, y_{\max}$, which define the top-left and bottom-right corners of the object's bounding box.

```
<annotation>
  <folder>Train</folder>
  <filename>01.png</filename>
  <path>/path/Train/01.png</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>224</width>
    <height>224</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>36</name>
    <pose>Frontal</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <occluded>0</occluded>
    <bndbox>
      <xmin>90</xmin>
      <xmax>190</xmax>
      <ymin>54</ymin>
      <ymax>70</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 4.5: Pascal VOC XML annotation file [29]

As shown in Figure 4.5, the PASCAL VOC format includes both required and optional fields. For object detection, the two main elements are the `name` and `bndbox` tags. Here, the former defines the class (e.g., `fire`, `smoke`, `human`, or `animal`) and the latter decides the box's coordinates. However, the annotation file also could be accompanied by more metadata such as `folder`, `filename`, and `path` that could serve for the organization of the dataset but are not required for model training. The same metadata fields like these can be further expanded to the optional key-value pairs to illustrate the possible values under each `object` tag of the `pose`, `truncated`, `difficult`, and `occluded`. This practice serves the purpose of specifying object visibility and providing the context of the implementation, thereby being easily neglectable if most object detection techniques are used. Nevertheless, it is an add-on that could help in the implementation of sophisticated training strategies or in the examination of the model's robustness while it is being occluded or its parts are being truncated.

4.2.3 Data Gathering and Categorization

To construct two comprehensive and diverse datasets, one for *RGB* imagery and one for *thermal* imagery, publicly available data was collected from various sources including GitHub, Roboflow, Zenodo, IEEE Dataport and many more. The initial data was categorized into four groups based on modality and content:

1. Images containing fire and potentially smoke in RGB imagery (see Table 4.1)
2. Images containing human and potentially animal in thermal imagery (see Table 4.2)
3. Images containing fire and potentially human in thermal imagery (see Table 4.3)
4. Images containing animal and potentially human in thermal imagery (see Table 4.4)

This categorization was used to facilitate efficient filtering, relabeling, and merging of datasets with common characteristics. Once proper processing and checking had been completed, the required data was integrated into the two final datasets that are RGB for the fire and smoke detection and thermal for the fire, human, and animal detection in infrared imagery.

Before presenting the datasets, it is important to clarify the annotation format strategy used in this thesis. All raw datasets were originally provided in either YOLO or PASCAL VOC format. During preprocessing, each dataset was converted to both formats to support training across various detection architectures.

The COCO format, however, was not generated at the per-dataset level. Instead, COCO-style annotations were created only after the final RGB and thermal datasets were formed. Thus, three separate JSON files for each of the train, validation, and test steps were generated. This step creates a format that is compatible with models such as EfficientDet that need COCO annotations for their execution.

Table 4.1: RGB – Fire/Smoke Datasets

Dataset Source	Original Format	Preprocessing Summary	Classes
DFireDataset [31]	YOLO	Test/val split, format check, filtering	Fire, Smoke
fire Dataset [32]	YOLO	Label correction (inverted IDs)	Fire, Smoke
synthetic fire-smoke Dataset [33]	YOLO	Direct format conversion	Fire, Smoke
FireMan-UAV-RGBT [34]	XML INFO	Cropping, annotation conversion, image filtering	Fire, Smoke
FlameVision [35]	YOLO	Manual labeling (smoke) with Roboflow, fire cleanup	Fire, Smoke

As shown in Table 4.1, five publicly available RGB datasets were gathered and preprocessed for the fire and smoke detection task. All datasets were initially annotated in YOLO format, except for FireMan-UAV-RGBT [34], which was provided in XML format. The DFireDataset [31] was converted to PASCAL VOC format, followed by a structured split from initial test set into validation and test sets. A subset of images was removed based on quality or annotation issues. The fire dataset [32] underwent label ID correction, as fire and smoke class labels were initially inverted. The synthetic fire-smoke dataset [33], generated using Unreal Engine, was directly converted from YOLO to PASCAL VOC format without additional modifications.

The FireMan-UAV-RGBT [34] annotations were converted to both YOLO and PASCAL VOC formats, and the images were cropped around either the smoke or fire bounding boxes depending on presence. Cropping was essential in this dataset because the original image resolution was Full HD or 4K, and the fire labels appeared very small within the frame. Without cropping, the model would struggle to learn meaningful features due to the limited pixel coverage of the fire object in distant, wide-area imagery. A fixed 640×640 crop was applied around the relevant region to ensure the fire was centered and better represented during training. Several images were excluded either due to annotation inconsistencies, the absence of detectable fire or smoke, or because they caused the model to struggle when mixed with im-

ages from other datasets. These conflicting samples introduced domain-level inconsistencies or label ambiguity, which negatively impacted the model's learning process and overall generalization. Finally, the FlameVision [35] dataset included RGB images with fire labels and required manual annotation for smoke classes using Roboflow. Some fire annotations were refined or removed before converting the final labeled dataset to PASCAL VOC format.

Table 4.2: Thermal – Human/Animal Datasets

Dataset Source	Original Format	Preprocessing Summary	Classes
Thermal Human UAV Dataset [36]	YOLO	Direct format conversion	Human
Thermal People Dataset [37]	YOLO	Manual labeling, relabeling	Human, Animal
Project FieldEye Dataset [38]	YOLO	Direct format conversion	Human
SAR-Train-O Dataset [39]	YOLO	Rearranged train/val/test splits	Human
BIRDSAI [40]	MOT CSV	Format conversion, filtering, train/val/test split	Human, Animal

As shown in Table 4.2, five publicly available thermal datasets were collected and preprocessed for detecting humans and potentially animal presence in thermal imagery. All datasets were originally annotated in YOLO format, except for BIRDSAI [40], which was provided in MOT CSV format. It was converted to PASCAL VOC using a custom converter. The data was also filtered for relevance and split into train, validation, and test subsets. Only samples with clear annotations for humans or animals were retained for training. The Thermal Human UAV dataset [36] and the Project FieldEye dataset [38] were directly converted to PASCAL VOC format without further modifications. The Thermal People dataset [37], obtained from Figshare and annotated using Roboflow, included both human and animal labels. It was converted to PASCAL VOC format after relabeling and quality verification. Finally, the SAR-Train-O dataset [39] required additional preprocessing: some validation and test images were originally included in the training set and were manually rearranged into proper

train/val/test splits prior to conversion.

The *MOT CSV* format is a frame-based annotation format commonly used in multi-object tracking tasks. Each row represents a detected object in a specific video frame and includes information such as the object's bounding box coordinates, object ID, and optionally class or visibility. Unlike VOC or COCO, it does not store image metadata or structured annotation hierarchies, making it lightweight but less suitable for standard object detection pipelines without conversion.

Table 4.3: Thermal – Fire/Human Datasets

Dataset Source	Original Format	Preprocessing Summary	Classes
Thermal Fire Dataset [41]	YOLO	Grayscale conversion, removal of irrelevant images	Fire
RGB-Thermal Wildfire dataset [42]	YOLO	Datasets merging and labeling	Fire, Human
FIReStereo [43]	YOLO	Frame extraction, labeling, grayscale processing	Fire
infrared-fire Dataset [44]	YOLO	Grayscale conversion only	Fire
FireMan-UAV-RGBT [34]	XML INFO	Grayscale and format conversion	Fire

As shown in Table 4.3, five publicly available thermal datasets were collected and preprocessed for detecting fire and potentially human presence in thermal imagery. All datasets were originally in YOLO format, except for FireMan-UAV-RGBT [34], which was provided in XML-based annotations. The Thermal Fire dataset [41] underwent grayscale conversion, and irrelevant images were removed to improve label consistency. The RGB-Thermal Wildfire dataset [42] was auto-labeled and manually corrected using Roboflow. It includes annotated samples for both fire and human presence and was created by combining images from the RGB-Thermal Wildfire dataset and a subset of the FIReStereo dataset [43].

The FIReStereo dataset [43] was extended by incorporating additional images and was annotated through a combination of auto-labeling and manual correction using Roboflow. It also includes samples from the FLAME dataset [45], which were extracted using a cus-

tom script for frame sampling. The merged dataset was labeled for infrared fire, downloaded in YOLO format, and subsequently converted to PASCAL VOC format. The infrared-fire dataset [44], derived from the FLAME-2 dataset [46], involved only grayscale conversion. Finally, the FireMan-UAV-RGBT dataset [34], originally in XML format, was converted to YOLO and PASCAL VOC formats and images were converted to grayscale to match the thermal modality.

Table 4.4: Thermal – Animal/Human Datasets

Dataset Source	Original Format	Preprocessing Summary	Classes
Wolf Dataset [47]	PASCAL VOC	Polygon-to-box conversion, label harmonization	Animal
Thermal Deer Dataset [48]	YOLO	Direct format conversion	Animal
Thermal Animals Dataset [49]	YOLO	Removal of irrelevant images	Animal
Thermal Animals Dataset [50]	YOLO	Direct format conversion	Animal, Human
Thermal Animals Dataset [51]	YOLO	Direct format conversion	Animal
Thermal Deer Dataset [52]	YOLO	Direct format conversion	Animal

As shown in Table 4.4, six publicly available thermal datasets were collected and preprocessed for detecting animals and potentially human presence in thermal imagery. All datasets were originally provided in YOLO format, except for the Wolf Dataset [47], which was downloaded in PASCAL VOC format. In this case, polygonal annotations were converted to rectangular bounding boxes, and class labels were harmonized to ensure consistency with the other datasets. The Thermal Deer Datasets [48], [52] were directly converted from YOLO to PASCAL VOC format without additional modification. Among the Thermal Animals Datasets [49], [50] and [51], only the dataset [49] was filtered to remove irrelevant or low-quality images prior to conversion. All three datasets were directly converted from YOLO to PASCAL VOC format to ensure annotation consistency and compatibility with the

overall training pipeline.

Across all datasets presented in Tables 4.1, 4.2, 4.3 and 4.4, bounding box annotations were reviewed for integrity, and image-level metadata was standardized to ensure compatibility across formats. Furthermore, the final combined RGB and thermal datasets were also exported in COCO format to facilitate training on models requiring JSON-based structured annotations. This preprocessing pipeline allowed for uniformity in format, consistency in labeling, and compatibility across different object detection architectures.

4.2.4 Applied Data Augmentation Techniques

Data augmentation stages followed two different approaches that correspond to the imaging modalities. The modes are termed as RGB and thermal, respectively. The typical image augmentations that were applied to RGB images include directly altering the color of the three channels. Thermal data, in contrast, retained their native single-channel grayscale format during augmentation to preserve the semantic meaning of intensity values. These changes were managed by using the Albumentations library, which is known for its suitability and high-speed nature in computer vision pipelines.

For better understanding of the augmentation effects, two exemplary snapshots, one RGB and one thermal, are displayed in Figures 4.6 and 4.7. The basic approach entails that some operations are used in both modalities while the others are specific to one of them (e.g., hue-saturation adjustments for RGB or salt-and-pepper noise for thermal). The majority of the mission is aimed at enhancing the model's resistance to distortions by integrating various types of noise that may be introduced due to weather or the sensor devices in the final deployment phase at real-world wildfire monitoring operations.

Figure 4.6: RGB frame

Figure 4.7: IR frame

The following augmentation techniques were applied to the training images to increase variability and robustness of the models. Each technique is selectively applied to either RGB or thermal data based on its relevance and visual semantics.

1. *Random Brightness Contrast:* Adjusts the brightness and contrast of RGB images. Brightness is randomly increased or decreased by up to 10%, and contrast is similarly adjusted by 10%, resulting in images that appear lighter or darker than the original. *Applies to RGB only.*

Figure 4.8: Random brightness contrast (RGB)

2. *Hue Saturation Value:* Modifies the hue, saturation, and value (brightness) of RGB images. Saturation may increase by up to 10 to make colors more vivid, and value can be shifted by up to 20 to alter overall image intensity. This augmentation is excluded for thermal images, as hue and saturation have no semantic meaning in grayscale imagery. *Applies to RGB only.*

Figure 4.9: Hue saturation value (RGB)

3. *Random Gamma:* Randomly adjusts the gamma of RGB images to simulate brightness variations. Gamma values between 90 and 110 result in images that are slightly more or less contrasted, aiding in generalization to real-world lighting conditions. *Applies to RGB only.*

Figure 4.10: Random Gamma (RGB)

4. *CLAHE (Contrast Limited Adaptive Histogram Equalization):* Enhances contrast in localized regions of RGB images, making low-light or low-contrast areas more visually distinct. This is especially useful for revealing detail in dark scenes. Although CLAHE can be beneficial for thermal images, in this work it is restricted to RGB data. *Applies to RGB only.*

Figure 4.11: CLAHE (RGB)

5. *Motion Blur:* Simulates motion artifacts common in UAV operations. A blur kernel of

size 3 to 5 pixels is randomly applied, producing streaking effects that replicate UAV vibration or fast movement. *Applies to both RGB and thermal/IR data.*

Figure 4.12: Motion blur (RGB)

Figure 4.13: Motion blur (IR)

6. *Horizontal Flip:* Flips the image horizontally, turning left-facing objects into right-facing ones. This augmentation improves invariance to orientation and is suitable for both RGB and thermal modalities. *Applies to both RGB and thermal/IR data.*

Figure 4.14: Horizontal flip (RGB)

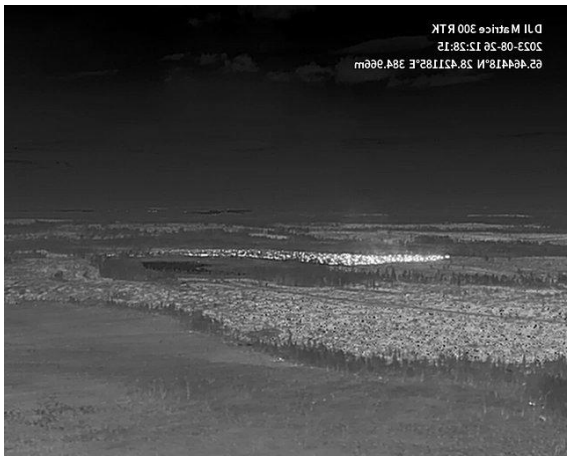


Figure 4.15: Horizontal flip (IR)

7. *Salt and Pepper Noise:* Introduces random black (pepper) and white (salt) pixels into thermal images, simulating sensor noise or transmission artifacts typical in aerial surveillance scenarios. The noise level is randomly chosen between 1% and 2% of total pixels. *Applies to thermal/IR only.*

Figure 4.16: Salt and pepper noise (IR)

8. *Affine Transformations:* A composite transformation that includes:

   Rotate: Randomly rotates the image within a range of $[-10°, 10°]$ or $[-15°, 15°]$, simulating camera tilt and perspective variation.

   Shear: Applies a shear transformation between $-5°$ and $5°$, producing diagonal distortion.

   Scale: Scales the image by a factor between 0.9 and 1.1, mimicking zoom in/out behavior.

   Translate: Translates the image horizontally and vertically by up to 10% of its dimensions, emulating camera displacement.

   These transformations collectively improve robustness to spatial deformations. *Applies to both RGB and thermal/IR data.*

Figure 4.17: Affine transformations (RGB)

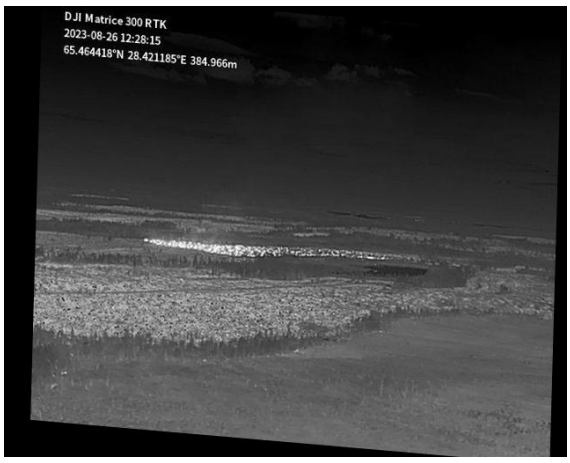


Figure 4.18: Affine transformations (IR)

For RGB images, two augmentations are randomly selected from a pool that includes random brightness contrast, hue saturation value, random gamma, CLAHE and motion blur. These are followed by a horizontal flip and affine transformations with 50% probability. For thermal/IR images, three augmentations are randomly selected and applied together from a set that includes horizontal flip, affine transformations, motion blur and salt-and-pepper noise in each frame. With this combination of augmentations, each modality undergoes distinct but carefully designed transformations that introduce controlled variability, enhancing the diversity of the dataset.

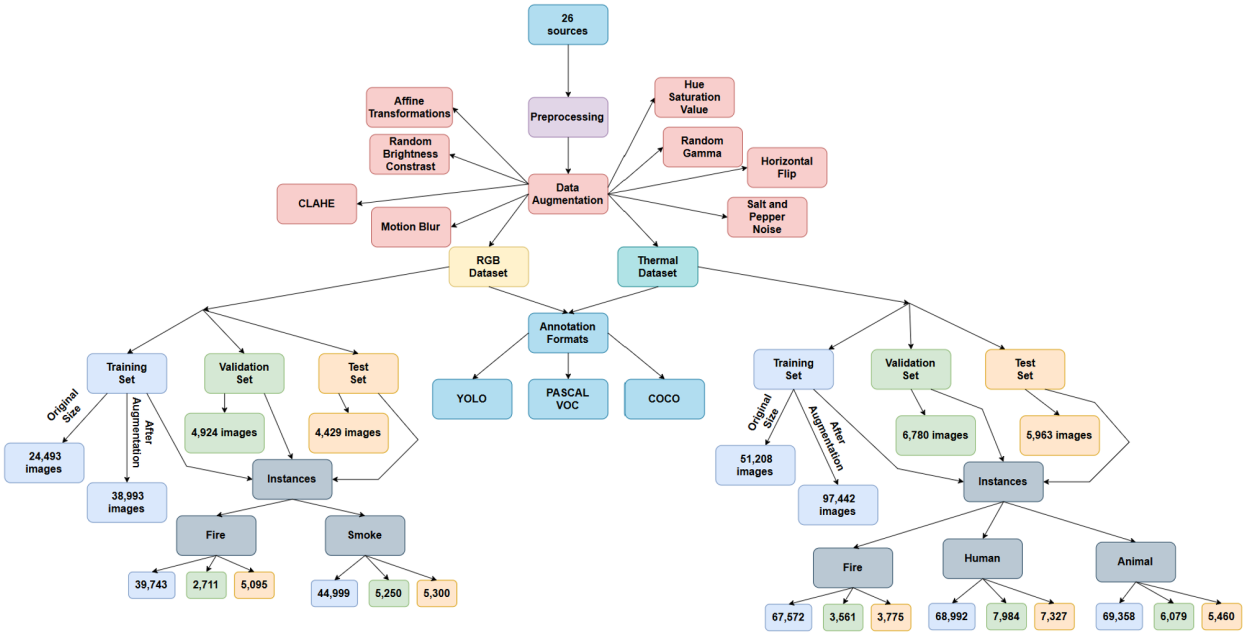


Figure 4.19: Datasets construction and augmentation pipeline

4.2.5 Final Datasets Composition

4.2.5.1 RGB Dataset

The RGB dataset was divided into three subsets: training, validation, and test. It comprises two annotated classes, *fire* and *smoke*, each accompanied by corresponding bounding boxes. Prior to augmentation, the dataset was partitioned into training, validation, and test sets, with *10%–15% of the original dataset allocated to validation and test*.

Before the application of data augmentation techniques, the training set consisted of *24,493 images*. Following augmentation, the number of training samples increased to *38,993 images*, comprising *39,743 instances of fire* and *44,999 instances of smoke* as shown in Fig-

ure 4.20. Although the dataset is slightly imbalanced, this is considered acceptable given that *smoke is a more common phenomenon in wildfires*.

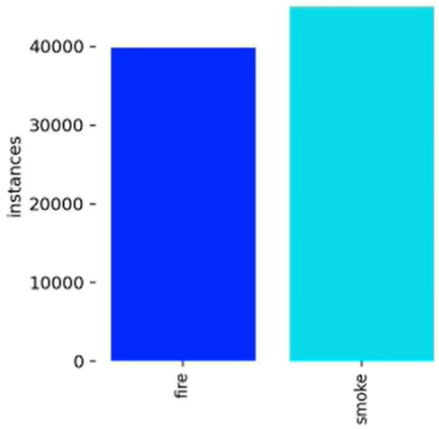


Figure 4.20: Distribution of instances in the RGB dataset

The *validation dataset* consists of *4,924 images*, containing *2,711 instances of fire* and *5,250 instances of smoke*.

The *test dataset* includes *4,429 images*, with *5,095 instances of fire* and *5,300 instances of smoke*.

The following Figure 4.21 presents representative samples from the combined datasets. *Validation* and *test* images are specifically showcased, as they are utilized in the Subsection 4.4.2 to demonstrate the performance of the best-performing model in detecting the various classes.

Figure 4.21: Samples from the RGB dataset

4.2.5.2 Thermal Dataset

The thermal dataset was divided into three subsets: training, validation, and test. It comprises three annotated classes, *fire*, *human*, and *animal*, each accompanied by corresponding bounding boxes. Prior to augmentation, the dataset was partitioned into training, validation, and test sets, with *10%–15% of the original dataset allocated to validation and test*.

Before the application of data augmentation techniques, the training set consisted of *51,208 images*. Following augmentation, the number of training samples increased to *97,422 images*, comprising *67,572 instances of fire*, *68,992 instances of humans*, and *69,358 instances of animals* as shown in Figure 4.22.

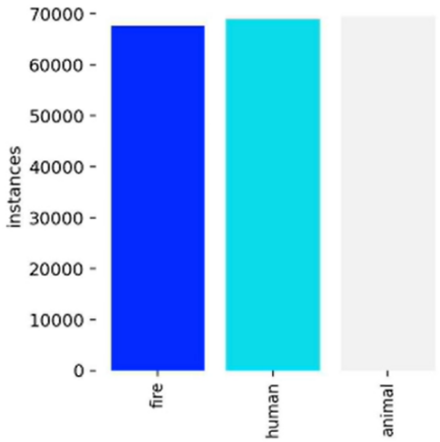
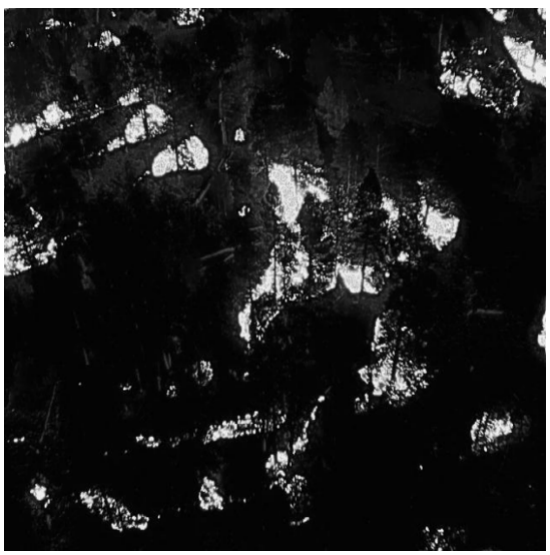


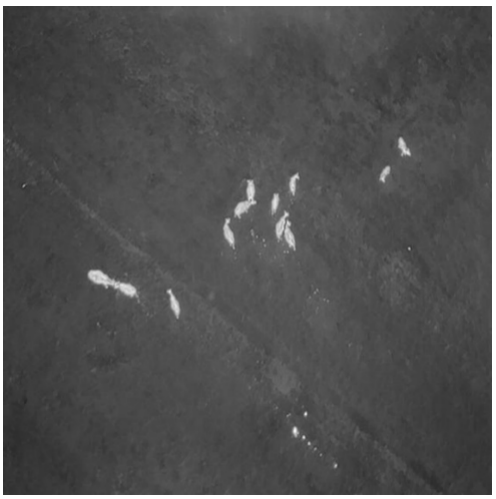
Figure 4.22: Distribution of instances in the Thermal dataset

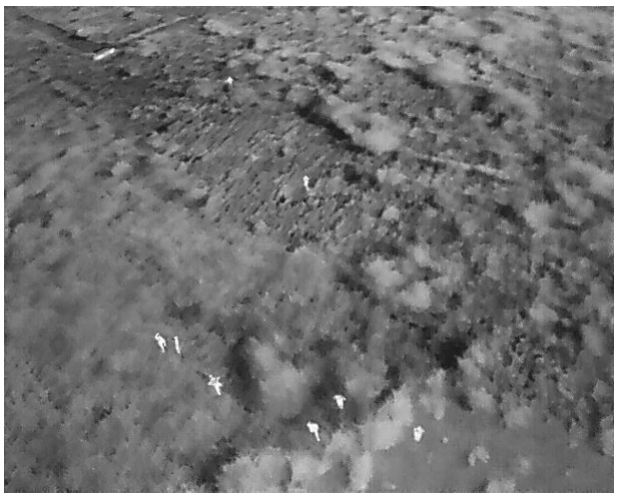
The *validation dataset* consists of *6,780 images*, containing *3,561 instances of fire*, *7,984 instances of humans*, and *6,079 instances of animals*.

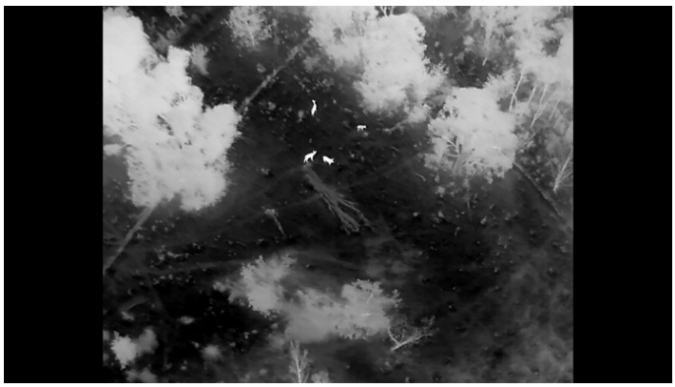
The *test dataset* includes *5,963 images*, with *3,775 instances of fire*, *7,327 instances of humans*, and *5,460 instances of animals*.

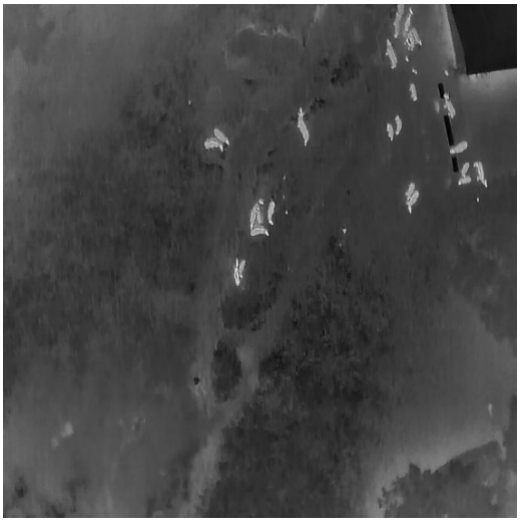
The following Figure 4.23 presents representative samples from the combined datasets. *Validation* and *test* images are specifically showcased, as they are utilized in the Subsection 4.4.3 to demonstrate the performance of the best-performing model in detecting the various classes.

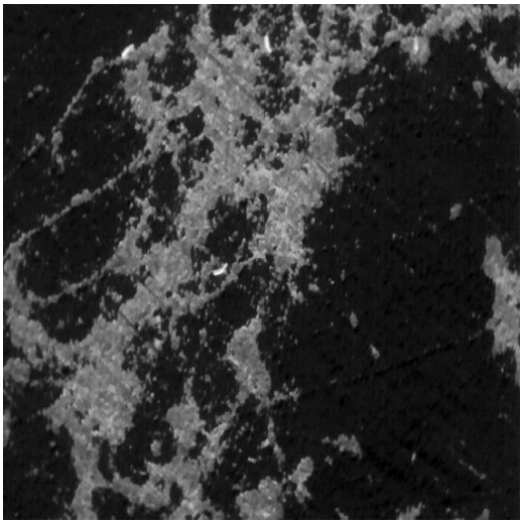


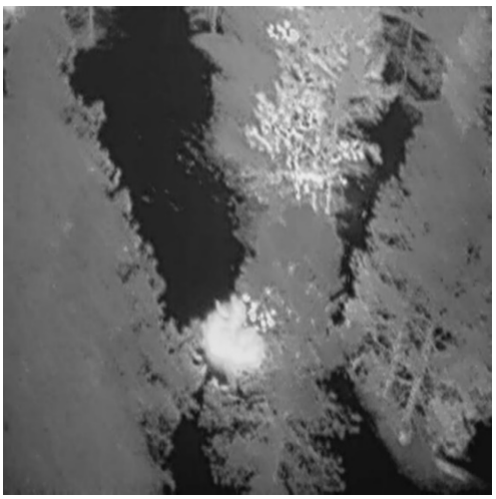




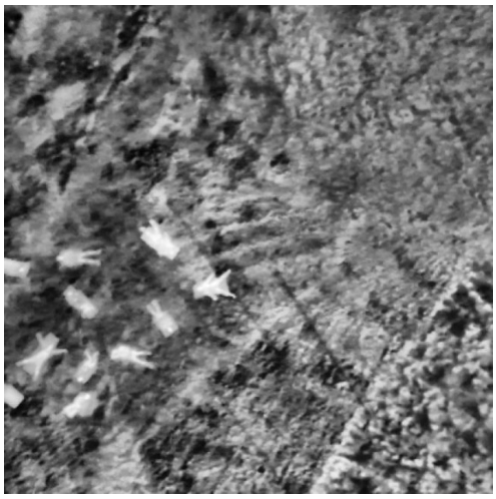


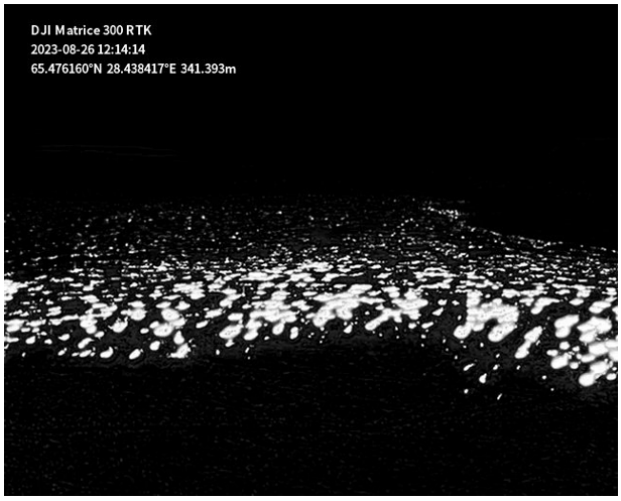







DJI Matrice 300 RTK
2023-08-26 12:28:37
65.467729°N 28.420457°E 385.006m




DJI Matrice 300 RTK
2023-08-26 12:14:14
65.476160°N 28.438417°E 341.393m

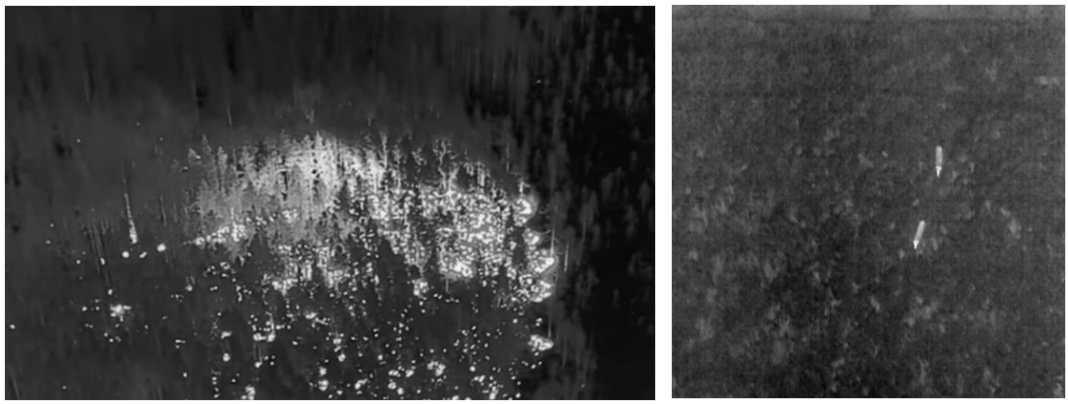


Figure 4.23: Samples from the Thermal dataset

4.3 Training Process

4.3.1 Introduction

The training process constitutes a critical phase in the development of a reliable and efficient machine learning-based wildfire detection system. In this chapter, the focus is placed on the methodologies, configurations, and evaluation strategies used to train the object detection models employed in this research. The objective is to make sure that the models not only acquire correct representations of the target classes, fire, smoke, human, and animal, but also generalize effectively to unseen data in real-world wildfire monitoring scenarios.

At the outset, the chapter begins by introducing the selected object detection models and elaborating on the specific loss functions and learning objectives each architecture optimizes during training. This is followed by a comprehensive study of their particular loss parts, the minds behind those losses, and the way in which they steer the model to a better classification and localization.

In addition, the metrics for evaluation are disclosed, which are relied upon to observe the training process and decide on the efficiency of the model. They embrace not only traditional classification-based indicators such as precision and recall but also those which deal especially with the detection field like Intersection over Union (IoU) and mean Average Precision (mAP). These metrics guides the choice of model and the scheduling of hyperparameters.

Some of the crucial procedures to improve the model such as use of stochastic gradient descent (SGD) with momentum and weight decay have been explained, and the schedule of the learning rate has been discussed using techniques like cosine annealing and warmup. In

order to secure model generalization and to prevent overfitting, the early stopping technique and a customized fitness score are two options. The latter provides an overall assessment target by incorporating numerous evaluation indicators at the same time for the best checkpoint during training.

This chapter outlines the training environment specs used for the experiments which cover high-performance desktop along with embedded platforms. Finally, a comprehensive comparison of model complexities, resource requirements, training times, and real-time inference performance is presented to justify the selection of the final models for both RGB and thermal detection tasks.

By presenting this comprehensive training pipeline, the chapter establishes the foundation for a fair, reproducible, and performance-aware evaluation of object detection architectures in the context of wildfire detection using UAV imagery.

4.3.2 Background

4.3.2.1 Machine Learning Models for Object Detection

The models used for training and comparison are listed and displayed below.

1. YOLO, Specifically YOLOv8s

YOLO (You Only Look Once) [53] implements a single, streamlined architecture for object detection. This method sees the detection task as a regression problem, thus it immediately estimates bounding boxes and class probabilities from only one evaluation. In contrast to the traditional methods like R-CNN, which separate region proposals and classification into different steps, YOLO is an end-to-end system that is optimized for real-time performance. YOLO uses a simplified convolutional architecture inspired by GoogLeNet. The architecture consists of a) *24 convolutional layers* for feature extraction, b) *2 fully connected layers* for predictions.

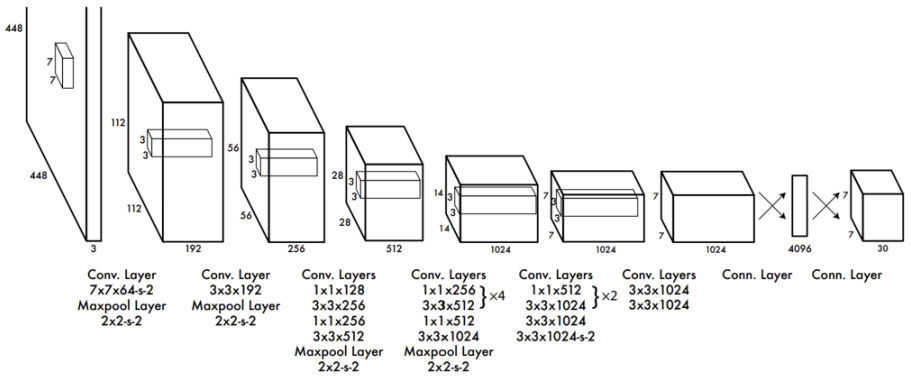


Figure 4.24: YOLO architecture [53]

YOLO uses only one convolutional neural network (CNN) that is able to cover the whole image and ensure a global understanding of the scene. The original image is divided into a grid, for example, SxS (such as 7x7 in YOLOv1). Then, each grid cell is in charge of detecting the objects, whose centers are located in that cell. Each cell predicts B bounding boxes (e.g., B=2) and associated confidence scores, along with C class probabilities conditioned on the presence of an object.

Bounding boxes are parameterized by x, y (center coordinates), w, h (width and height, normalized by image dimensions), and a confidence score that combines the likelihood of object presence with the accuracy of the bounding box. The YOLO loss function integrates:

- *Localization Loss:* Penalizes errors in x, y, w, h.
- *Confidence Loss:* Measures the difference between predicted and actual confidence scores.
- *Class Probability Loss:* Quantifies errors in predicted class probabilities.

YOLOv8 Overview

YOLOv8 [54], [55], [56] is very recent addition to the YOLO family, it is intended for object detection, image classification, and instance segmentation. Ultralytics is the developer of YOLOv8 that takes into consideration the positive aspects of YOLOv5 and at the same time incorporates major changes in both architecture and training techniques, thus boosting the performance, accuracy, and efficiency for tasks of computer vision nowadays.

Key Innovations in YOLOv8

1. *Improved Backbone (CSPNet):* Enhances efficiency and accuracy compared to earlier YOLO versions.
2. *PANet Head:* Improves robustness against occlusions and scale variations in objects.
3. *Hybrid Training:* Combines supervised and unsupervised learning for better data utilization.
4. *Anchor-Free Detection:* Predicts bounding box centers directly, eliminating the need for predefined anchor boxes. This improves robustness and reduces computational complexity, speeding up post-processing (e.g., Non-Maximum Suppression).
5. *Training Tricks:*
    - *Mosaic Augmentation:* Combines multiple images into one for training, increasing data diversity. (Disabled in this thesis for fair comparison with other models.)
    - *Decoupled Head:* Separates classification and bounding box regression into distinct branches within the detection head, improving accuracy and efficiency.

These innovations position YOLOv8 as a cutting-edge model for real-time object detection, offering superior accuracy and faster processing.

Yolov8 Model Selection

Among various model sizes such as nano, small, medium, large, and extra large, the small one is opted in this research for achieving a balanced compromise between accuracy and inference speed. Larger models are always known to be more precise, but they also need significantly increased computational power and cause high latency, which is not suitable for real-time applications. On the other hand, small models still have accuracy comparable to large ones, but the model size and the inference time are drastically reduced. The nano model, although even faster and lighter, are left out due to their conspicuously lower detection accuracy, which leads to unreliability. Hence, the small models are chosen as the best fit for time-sensitive wildfire detection tasks, where fast and reliable performance is necessary. This also makes them most suitable for time-sensitive wildfire detection tasks, where quick and efficient decision-making is vital.

Implementation Details

- *Training Framework:* The training process leverages the Ultralytics Python library for seamless implementation.
- *Metrics Logging:* Metrics are tracked with MLflow.
- *Extended Metrics:* Custom modifications were made to the Ultralytics library to include mAP@75 and AP@75 for each class.

Annotation Format

- The YOLO annotation format is used for training, alongside a `data.yaml` file that defines dataset configurations and training parameters.

2. Faster R-CNN with MobileNetV3 Backbone

MobileNetV3 was selected as the backbone for Faster R-CNN instead of ResNet50 due to its lower computational requirements. ResNet50 exceeds 100 GFLOPS, making it unsuitable for edge devices with limited resources. MobileNetV3 introduces significant innovations that enhance both efficiency and accuracy, rendering it suitable for mobile and edge deployment scenarios.

MobileNetV3 Innovations

MobileNetV3 [57] incorporates *Squeeze-and-Excitation (SE) modules* along with *inverted residual blocks* to facilitate the increase in network efficiency. SE modules energize the network's attention to compelling features by dynamically tuning the representativeness of each channel, thus adding more representational power with minimal extra computational cost. The inverted residual blocks, taking over from MobileNetV2, apply a *linear bottleneck design* that enables no loss of information by avoiding the non-linearities at the bottleneck outputs.

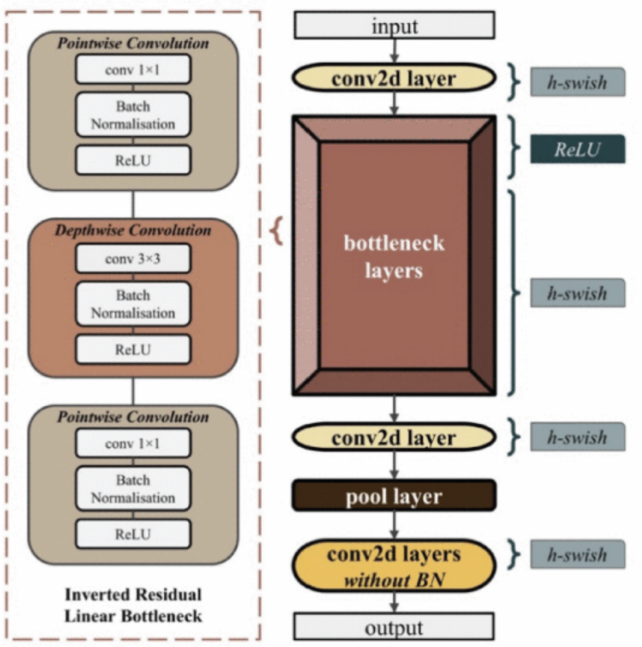


Figure 4.25: MobileNetV3 architecture [58]

The foremost breakthrough of MobileNetV3 is that it embraces the *Neural Architecture Search (NAS)* for the purpose of searching the best structural parameters of the model which will be then implemented in the edge devices. NAS finds the best possible combination of the layers, widths, and operations that fits perfectly with the efficiency needed. Besides that, MobileNetV3 uses the *h-swish activation function*, which is a less costly version of swish and still provides almost the same accuracy with less complexity.

MobileNetV3 is available in two variants:

- *MobileNetV3-Large*: Suited for applications that demand high accuracy.
- *MobileNetV3-Small*: Intended for environments with limited resources. Both versions incorporate a *Lite Reduced Atrous Spatial Pyramid Pooling (LR-ASPP)* module for improved segmentation performance.

MobileNetV3 provides enhanced classification accuracy, reduced latency, and robust performance in detection and segmentation, making it appropriate for diverse edge-focused applications.

Faster R-CNN Architecture

Faster R-CNN [59] is structured around two primary components, *Region Proposal Networks (RPNs)* and *Fast R-CNN*, which form a unified, end-to-end trainable framework for object detection. The core components are outlined below:

1. *Feature Extraction*: Begins with a convolutional backbone (e.g., MobileNetV3) to derive feature maps from input images. These are shared between the RPN and detection head to reduce computational load.

2. *Region Proposal Network (RPN)*: A fully convolutional component that generates *region proposals*. It applies a small sliding window over the feature map to predict k anchor boxes. Each anchor outputs:

    - An *objectness score* indicating whether the anchor is foreground or background.
    - *Refined bounding box coordinates*, with redundant proposals filtered via *Non-Maximum Suppression (NMS)*.

3. *ROI Pooling*: Converts proposals into fixed-size feature maps compatible with the detection head.

4. *Fast R-CNN Detection Head*: Processes fixed-size features to classify object categories and refine bounding box positions.

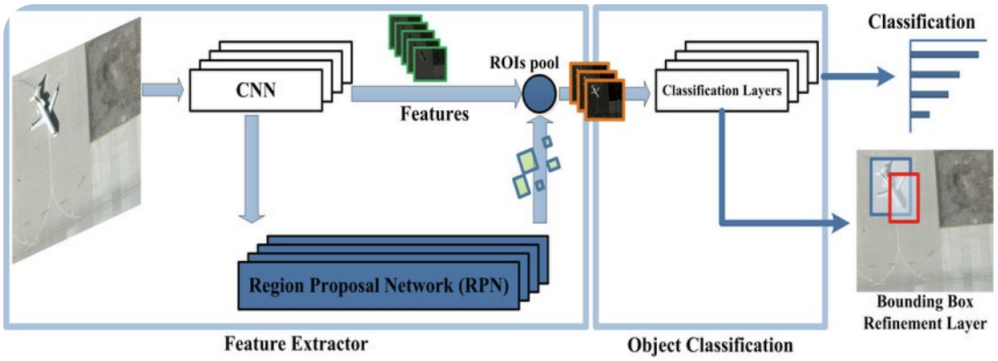


Figure 4.26: Faster R-CNN architecture [60]

Faster R-CNN operates as a *two-stage detector*:

- *Stage 1*: RPN generates region proposals.
- *Stage 2*: The detection head classifies and refines proposals.

Implementation Details

Our project was accomplished by utilizing `fasterrcnn_mobilenet_v3_large_fpn`, a model from the PyTorch Vision library. The dataset annotations correspond to *PASCAL VOC format* style, where bounding box coordinates are indicated with XML files. The parameters of the model are gotten through a specifically designed script, which makes it possible to perform a standard evaluation of the object detection capability. MLflow makes sure that all outputs of evaluation are well organized and kept.

3. SSD with VGG16 Backbone

VGG16 Architecture

The VGG16 model [61], is a work of Oxford's Visual Geometry Group and is described in their paper *"Very Deep Convolutional Networks for Large-Scale Image Recognition"* utilizes smaller convolutional filters (3x3) to increase the network depth while keeping the computational costs at a low level.

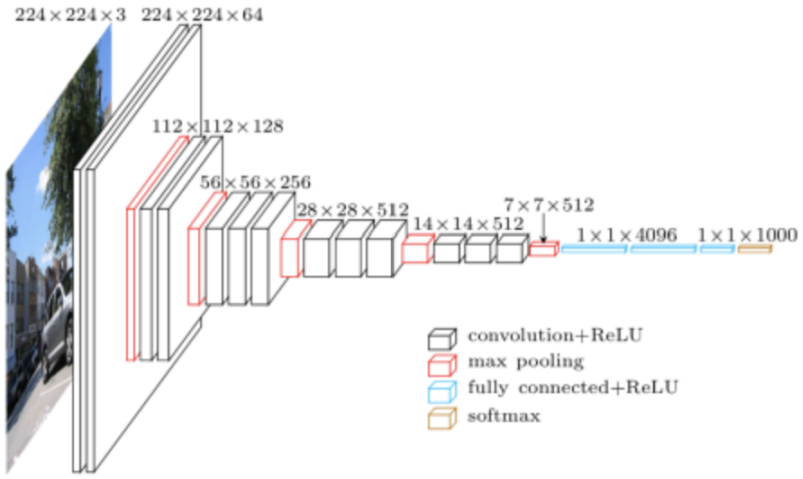


Figure 4.27: VGG16 architecture [62]

VGG16 consists of 16 layers as shown in Figure 4.27 with the following structure:

- *13 Convolutional Layers*: Use 3x3 filters for compact spatial representation.
- *Maxpool Layers*: Interleaved to reduce dimensions while retaining critical features.
- *3 Fully Connected Layers*: Responsible for final predictions.

This architecture balances simplicity and depth, contributing to VGG16's effectiveness in vision tasks.

SSD: Single Shot MultiBox Detector

The *"SSD: Single Shot MultiBox Detector"* framework [63] is designed for real-time object detection. Unlike two-stage approaches, SSD is a one-stage detector that makes predictions directly from feature maps.

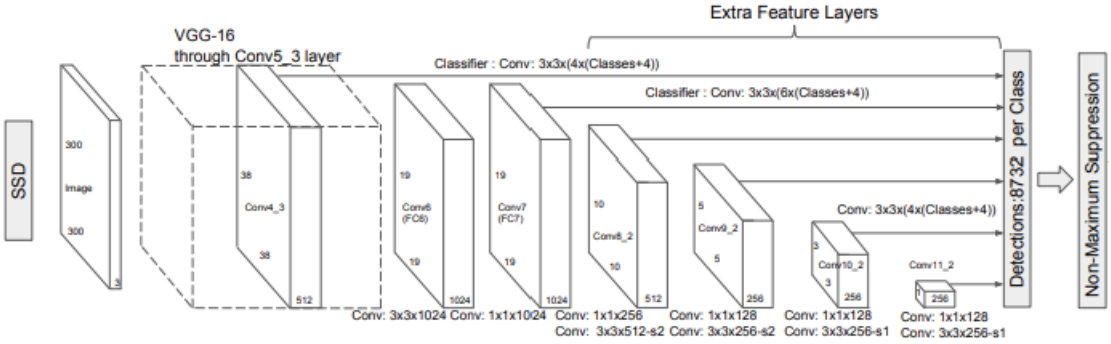


Figure 4.28: SSD architecture [63]

How SSD Works

1. *Single Network Architecture*: A single deep network processes the image end-to-end.
2. *Default Boxes*: Each grid cell in the image is assigned predefined bounding boxes of varying shapes and aspect ratios (priors).
3. *Multi-Scale Feature Maps*: Features from different layers support detection at multiple scales.
4. *Predictions*:
    - Class probabilities for each feature map cell.
    - Adjusted coordinates for bounding boxes.
5. *Training and Inference*:
    - During training, default boxes are matched to ground truth using Intersection over Union (IoU).
    - During inference, predictions are followed by NMS to filter duplicates.

Implementation Details

The `ssd300_vgg16` model from the PyTorch Vision library was employed to carry out the experiment. The dataset annotations match the *PASCAL VOC format* where the bounding box information is given in XML files. The performance is evaluated with mean Average Precision (mAP), Intersection over Union (IoU), by using a custom script. The evaluation results are recorded and observed trough MLflow.

4. EfficientDet D1

The EfficientDet family [64] is from the paper *"EfficientDet: Scalable and Efficient Object Detection"*, that highlights the efficiency aspect of the trade-off between detection accuracy and computing power which has been established over the diverse resource environments. EfficientDet utilizes the EfficientNet backbone, which is a network that has been designed to maximize depth, width, and resolution with minimum parameters, allowing effective feature extraction. Besides that, EfficientDet makes use of the *Bi-directional Feature Pyramid Network (BiFPN)* to improve multi-scale feature fusion. BiFPN exploits the weights that can be learned, so it can balance the importance of each layer and give the most efficient/accurate object detection at the output.

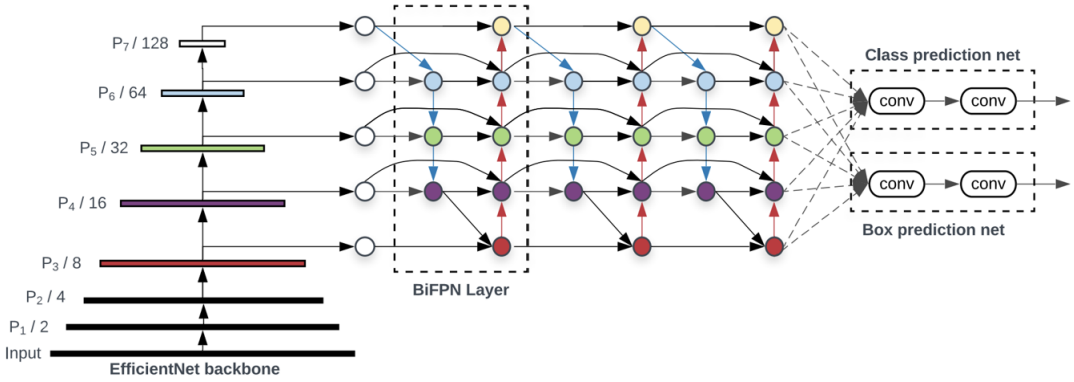


Figure 4.29: EfficientDet architecture [64]

How EfficientDet Works

1. *Feature Extraction*: The backbone EfficientNet-B1 is used to extract multi-scale feature representations. This version is lightweight and particularly suitable for low-resource scenarios.

2. *Feature Fusion with BiFPN*: The extracted features are passed to the BiFPN, which performs feature fusion across multiple levels using learnable weights. This allows the

network to emphasize the most informative features.

3. *Prediction Network*: The fused features are input to two distinct heads:

    - *Class prediction head*: Estimates the probability distribution over object categories.

    - *Box prediction head*: Refines the coordinates of predicted bounding boxes.

4. *Compound Scaling*: EfficientDet employs compound scaling to uniformly adjust depth, width, and resolution across the model, maintaining a balance between performance and efficiency tailored to resource constraints.

EfficientDet D1 Specifics

The D1 variant of EfficientDet employs EfficientNet-B1, a configuration that provides a favorable trade-off between computational cost and performance. It is particularly suited for applications that require efficient deployment on constrained hardware platforms.

Implementation Details

- *Library and Pretrained Models*: The implementation utilizes the `effdet` Python library, which includes pretrained EfficientDet weights.

- *Annotation Format*: Annotations follow the COCO JSON format, with bounding boxes formatted as $[x_{\text{max}}, x_{\text{min}}, y_{\text{max}}, y_{\text{min}}]$.

- *Metrics*: Model performance, including mean Average Precision (mAP), is evaluated using a custom script and the results are logged with MLflow.

4.3.2.2 Analysis of Loss Calculations Across All Models

During the training and validation of object detection models, losses are aggregated across multiple components depending on the model architecture. These losses are computed over the train/validation datasets and guide the optimization process. The following outlines the main loss components used by each model:

1. YOLOv8s (You Only Look Once)

- *box_loss:* This is the Complete Intersection over Union (CIOU) loss, a regression-based loss used to measure the alignment accuracy between predicted and ground truth

bounding boxes. CIOU considers positional, scale, and shape similarities, which contribute to more precise box predictions.

- *cls_loss:* This represents the classification loss, implemented as Variable Focal Loss (VFL). VFL improves upon standard Focal Loss by adjusting the weighting of positive and negative samples, making the model more focused on hard-to-classify instances and addressing class imbalance.

- *dfl_loss:* This is the Distribution Focal Loss (DFL), which helps enhance detection performance on small targets. DFL incorporates scale-aware information to better weigh hard examples during training.

$$\text{Total Loss}_{YOLOv8} = \text{cls\_loss} + \text{box\_loss} + \text{dfl\_loss}$$

2. Faster R-CNN

- *loss_classifier:* Classification loss calculated from Region of Interest (RoI) head predictions. After RoI proposals are generated, this loss evaluates the accuracy of category assignments.

- *loss_box_reg:* Bounding box regression loss for refining object coordinates within each RoI.

- *loss_objectness:* Objectness loss derived from the Region Proposal Network (RPN), evaluating how well anchor boxes are predicted as object-containing regions.

- *loss_rpn_box_reg:* Bounding box regression loss within the RPN, which adjusts anchor coordinates based on ground truth alignment.

$$\begin{aligned}\text{Total Loss}_{Faster\text{-}RCNN} = {} & \text{loss\_classifier} \\ & + \text{loss\_box\_reg} + \text{loss\_objectness} + \text{loss\_rpn\_box\_reg}\end{aligned}$$

3. SSD (Single Shot Multibox Detector)

- *classification:* The classification loss evaluates how accurately objects are assigned to the correct category.

- *bbox_regression:* The bounding box regression loss measures the localization accuracy of predicted boxes. A weight factor a (commonly set to 1) is applied to this component.

$$\text{Total Loss}_{SSD} = \text{classification} + a \cdot \text{bbox\_regression}, \quad \text{where } a = 1$$

4. EfficientDet-D1

- *loss:* This is the total loss used for optimization and consists of a weighted combination of classification and box regression losses.
  - *class_loss:* Object classification loss.
  - *box_loss:* Bounding box regression loss.

$$\begin{aligned}\text{Total Loss}_{EfficientDet} = \text{class\_loss} \\ +\text{box\_loss\_weight} \cdot \text{box\_loss}, \quad \text{where box\_loss\_weight} \approx 50\end{aligned}$$

4.3.2.3 Evaluation Metrics

Evaluating object detection models is a must-have requirement for their safety and reliability services in real-world scenarios. Several benchmarking metrics are involved to evaluate the detector's capability to detect the object and correctly localize it. The evaluation metrics are *Precision*, *Recall*, *F1 Score*, and *Mean Average Precision (mAP)* that help to learn detection accuracy, types of errors in terms of false positives and negatives, and overall reliability of the model.

An additional metric such as *Intersection over Union (IoU)* is also used to refine performance evaluations considering the spatial overlap between predicted and ground truth bounding boxes.

1. Intersection over Union (IoU)

IoU quantifies the overlap between predicted and ground truth bounding boxes. It is calculated as:

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

The IoU threshold determines whether a predicted box is a correct match to a ground truth box. Common thresholds are 0.5 (for mAP@0.5) or higher for stricter evaluation.

True Positives (TP): Predictions that correctly detect and localize an object with IoU above the threshold.

False Positives (FP): Predictions on non-existent or incorrectly localized objects (IoU below threshold).

False Negatives (FN): Ground truth objects missed by the model.

True Negatives (TN): Not applicable in object detection due to the nature of spatial predictions.

2. Precision

Precision is defined as the ratio of true positives (TP) to all predicted positive samples, which includes both true positives and false positives (FP). It measures the model's ability to avoid false alarms.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

Precision answers the question: "Of all the instances the model predicted as positive, how many were actually correct?"

3. Recall

Recall measures the model's ability to detect all relevant instances. It is the ratio of true positives to all actual positive instances, including false negatives (FN).

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

It answers the question: "Of all the actual positive cases, how many did the model correctly identify?"

4. F1 Score

The F1 score is the harmonic mean of precision and recall. It provides a balanced evaluation when precision and recall are in tension. It is computed as:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is especially useful in cases with class imbalance or when both false positives and false negatives are critical.

5. Mean Average Precision (mAP)

The mean Average Precision (mAP) is a comprehensive metric commonly used in object detection. It is computed as the mean of the average precisions (APs) across all classes. The steps to calculate mAP are:

i) *Precision–Recall Curve:* Precision and recall are plotted, and the area under the curve (AUC) gives the AP for each class.

ii) *Mean Aggregation:* The mAP is computed by averaging the APs across all object classes.

Various versions of mAP depend on the Intersection over Union (IoU) thresholds used:

- *mAP@0.5:* Evaluates detections correct if IoU ≥ 0.5 (PASCAL VOC-style).
- *mAP@0.75:* A stricter version with IoU ≥ 0.75.
- *mAP@0.5:0.95:* The COCO-style mAP, averaged over IoUs from 0.5 to 0.95 in 0.05 steps, offering a more robust and challenging evaluation.

4.3.2.4 Optimizer

SGD Overview

Stochastic Gradient Descent (SGD) is an optimization algorithm that updates model parameters using mini-batch gradients, making it computationally efficient for large datasets. The update rule is:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla L(\theta_t) \tag{4.1}$$

where η is the learning rate.

Momentum in SGD

Momentum helps accelerate convergence by smoothing updates using a moving average of past gradients:

$$v_{t+1} = \beta \cdot v_t + \nabla L(\theta_t) \tag{4.2}$$

$$\theta_{t+1} = \theta_t - \eta \cdot v_{t+1} \tag{4.3}$$

where:

- v_t is the velocity term,
- β is the momentum coefficient (typically 0.937).

Momentum helps overcome small gradient fluctuations, improving convergence speed and stability.

Weight Decay in SGD

Weight decay is a regularization technique that prevents overfitting by penalizing large weights. It is implemented as L2 regularization, modifying the SGD update as:

$$\theta_{t+1} = \theta_t - \eta \left(\nabla L(\theta_t) + \lambda \theta_t \right) \tag{4.4}$$

where λ is the weight decay coefficient.

SGD with Momentum and Weight Decay

The combined update rule for Stochastic Gradient Descent (SGD), incorporating both momentum and weight decay, is defined as:

$$v_{t+1} = \beta \cdot v_t + \nabla L(\theta_t) + \lambda \theta_t \tag{4.5}$$

$$\theta_{t+1} = \theta_t - \eta \cdot v_{t+1} \tag{4.6}$$

Advantages of Incorporating Momentum and Weight Decay into SGD

- Momentum accelerates convergence and stabilizes updates.
- Weight decay improves generalization by controlling the growth of model weights.
- Together, they balance training speed and model regularization.

In this thesis, SGD with momentum and weight decay is utilized to achieve faster convergence, smoother optimization dynamics, and improved generalization performance, particularly in the training of deep neural networks.

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay), μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

Figure 4.30: PyTorch implementation of the SGD optimizer [65]

4.3.2.5 Learning Rate Scheduling with Warmup

Cosine Decay Learning Rate Scheduling

Cosine decay [66] is a learning rate scheduling technique commonly used in training deep learning models. It dynamically adjusts the learning rate (LR) throughout the training process to improve convergence speed and potentially enhance model performance.

It should be noted that our training pipeline is a schedule that adjusts the learning rate of the model according to a cosine annealing cycle using the `CosineAnnealingLR` scheduler. This method is a gradual linear decrease of the learning rate from a certain maximal value to a minimal one over the specified number of epochs/iterations. It smooths and never breaks transitions that are very comfortable for such cases, where a gradual decline of the learning rate is required throughout the training process.

The update rule for the `CosineAnnealingLR` scheduler is given by:

$$\eta_t = \eta_{\text{min}} + \frac{1}{2} \cdot (\eta_{\text{max}} - \eta_{\text{min}}) \cdot \left(1 + \cos\left(\frac{T_{\text{cur}}}{T_{\text{max}}} \cdot \pi\right)\right) \tag{4.7}$$

where:

- η_t: The learning rate at the current step t.

- η_{min}: The minimum learning rate to which the scheduler will decay.
- η_{max}: The maximum learning rate at the start of the cycle.
- T_{cur}: The number of epochs elapsed since the beginning of the current cosine cycle.
- T_{max}: The total number of epochs in the cosine decay cycle.

Warmup Scheduling

The warmup period is a learning rate scheduling method wherein the learning rate slowly goes up from a low initial value to the target value in the first few epochs. Typically, it is combined with other strategies such as cosine decay to improve training stability and convergence.

What the warmup phase mainly aims at is to give the model an opportunity to search the parameter space with a smaller learning rate at the beginning of training. It has been shown that this lowers the loss function's large fluctuations and it is more feasible to achieve stable optimization during the initial stages of learning.

These models are Faster R-CNN, SSD, and EfficientDet.

When it comes to the YOLO-based models, the warmup phase is done in a different way. The training does not proceed starting with a low learning rate, but on the contrary, a high learning rate is used that is reduced gradually during the first three epochs until the target learning rate is reached. This approach has been revealed to be suitable for YOLO models, which cannot be easily disturbed by high learning rates in the initial training phase.

4.3.2.6 Early Stopping and Keeping the Best Model During Training

Early stopping is a regularization technique used during the training of neural networks to prevent overfitting. To fully understand the rationale behind early stopping, it is first essential to comprehend the concept of overfitting.

Definition of Overfitting

Overfitting [67] is a typical problem in machine and deep learning, wherein a model becomes highly familiar with training data, including noise and random fluctuations. Although such a model performs exceptionally well on training data, it often exhibits significantly reduced performance on unseen data (e.g., validation or test sets), indicating poor generalization.

Causes of Overfitting

1. *Overly Complex Model:* Models with an excessive number of parameters relative to the dataset size may memorize the training data, including noise, leading to reduced generalization.
2. *Insufficient Training Data:* A small dataset limits the model's ability to learn general patterns, increasing the risk of overfitting to training noise.
3. *Lack of Regularization:* Regularization techniques, such as L1 and L2 regularization, mitigate overfitting by penalizing large weights through constraints on the loss function.

Techniques for Preventing Overfitting

Several strategies are commonly employed to mitigate overfitting:

1. Increase training data
2. Apply regularization
3. Use dropout
4. Employ early stopping
5. Apply data augmentation

Early Stopping as a Regularization Strategy

Early stopping [67] prevents overfitting to some extent by assessing the model quality on the validation set and terminating usage if no further decrease in validation loss is detected in the last several epochs. This allows the model to become more general by not being excessively trained on the noise.

Mechanism of Early Stopping

1. *Validation Set:* A portion of the training dataset is set aside as a validation set to evaluate performance after each epoch.
2. *Performance Monitoring:* Performance on the validation set is tracked using metrics such as validation loss or accuracy. These metrics help assess the model's generalization capacity.

3. *Early Stopping Criterion:* Training is stopped when validation performance deteriorates (e.g., increased loss). A patience parameter allows for tolerating minor fluctuations before termination.

4. *Model Selection:* The model corresponding to the best validation performance is selected as the final version to avoid overfitting to training noise.

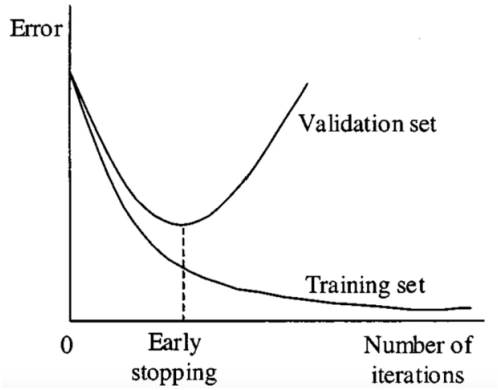


Figure 4.31: Early stopping mechanism [67]

Custom Fitness Score for Model Selection

To retain the best-performing model during training, we define a custom fitness score that combines multiple object detection metrics. This score ensures selection based on actual detection performance rather than raw loss values.

Definition of the Fitness Score

The custom fitness score is defined as:

$$\textit{Fitness score} = 0.1 \cdot mAP@50 + 0.9 \cdot mAP@50:95$$

This formulation places greater emphasis on *mAP@50:95*, a more comprehensive and stricter metric, while assigning a smaller weight to *mAP@50*, a more lenient criterion. This balanced weighting ensures the selected model performs robustly across various Intersection over Union (IoU) thresholds.

Advantages of the Fitness Score

This metric is quite useful for avoiding pitfall where the only decision is made based on the validation loss which might not be an ideal indicator of detection quality. Although validation loss can vary or go up without any significant change in real-world application, numbers like *mean Average Precision (mAP)* are a more direct way of measuring detection

accuracy. To give an example, the loss on a validation set can go up during *7 consecutive epochs* without a decrease in detection performance. Therefore, the fitness score gives a more dependable ground for choosing a model.

4.3.3 Training Environment Specifications

The training experiments conducted in this thesis were executed on a high-performance computational system with the following specifications:

- *Processor:* Intel Core i9-13900KF, a 13^{th}-generation processor featuring 24 cores and 32 threads, with a maximum clock speed of 5.8 GHz. This processor delivers exceptional computational capabilities for parallel processing tasks.

- *Graphics Processing Unit (GPU):* NVIDIA GeForce RTX 4090 equipped with 24 GB of VRAM. The GPU operates using CUDA version 12.4 to accelerate deep learning computations, facilitating efficient training of complex machine learning models.

- *Memory Architecture:* The system supports 39-bit physical and 48-bit virtual memory addressing and is equipped with 128 GB of RAM. This enables the seamless processing of large-scale datasets and supports memory-intensive operations.

- *Operating System:* Ubuntu 22.04 LTS (Linux), running on an `x86_64` architecture. This environment supports both 32-bit and 64-bit applications, ensuring compatibility with a broad range of software tools.

- *Deep Learning Framework:* PyTorch, which provides dynamic computation graphs, efficient GPU utilization, and extensive support for model development, training, and deployment.

This configuration provides a robust environment for conducting computational experiments by offering the necessary resources to train, validate, and test deep learning models effectively. The synergy between the powerful processor and the high-end GPU ensures efficient parallelism and accelerated model convergence.

4.3.4 Comparison of Model Complexity and Resource Requirements

4.3.4.1 Introduction

This section analyzes and compares the computational complexity and resource requirements of four widely used object detection models: *YOLOv8s*, *Faster R-CNN with MobileNet-V3*, *SSD with VGG16*, and *EfficientDet D1*. The decision is based on the key parameters that influence the practicability of implementation such as trainable parameters number, GFLOPs (Giga Floating Point Operations per second), and the total storage size of each model. These parameters are indispensable in choosing the right model for real-time applications, especially when one tries to find a good balance between the precision, computational efficiency, and memory usage.

4.3.4.2 Comparison

To evaluate the computational efficiency of each model, the following metrics are considered:

- *Number of Parameters (M):* Represents the total number of trainable parameters in the model, which directly impacts memory usage and inference speed.
- *GFLOPs:* Denotes the number of floating-point operations required per forward pass, serving as a measure of computational complexity.
- *Storage Size (MB):* Indicates the size of the saved model, relevant for scenarios with storage constraints.

Table 4.5: Comparison of models based on complexity and resource requirements

Model	Parameters (M)	GFLOPs	Storage Size (MB)
YOLOv8s	11.13	28.6	**21.5**
Faster R-CNN (MobileNetV3)	19.38	**4.49**	72.5
SSD (VGG16)	35.64	34.86	91.6
EfficientDet D1	**6.63**	6.1	26.8

4.3.4.3 Conclusions

The quantitative comparison of Table 4.5 provides a direct insight into the resource requirements and complexity of the various models.

EfficientDet D1 is the most lightweight and computationally efficient model as it is the least number of parameters, GFLOPs, and storage size that it has among all the candidates. Its minimal resource consumption makes it especially well-suited for usage in areas with limited computational and memory capacity such as embedded systems or mobile edge devices.

YOLOv8s covers all the spectrums evenly across the measured metrics. Being of a compact structure and with moderate needs in computation and storage it provides an optimal compromise between performance and efficiency. These features make it an appealing solution for real-time inference tasks, particularly in edge-based applications where both speed and resource efficiency are critical.

Faster R-CNN (MobileNetV3) has an average number of parameters and low computational cost in terms of GFLOPs. But it has a relatively big storage space that may cause it to be less suitable for situations where the available memory is limited, thus reducing its suitability for highly resource-constrained platforms.

On the other hand, **SSD (VGG16)** is the heaviest model that consumes the most resources. It has the highest number of parameters and the largest memory size which limits its practical use to the systems that have, are compatible with, or can share of the computational and memory power.

The results clearly explain the compromises faced in the process of selecting a model, showing how different designs can be different in their computational complexity and ability to be deployed. Thus, this study lays out a comprehensive framework for deciding the most suitable model by taking into account the exact limitations and goals of the chosen application.

4.3.5 Real-Time Inference Performance Evaluation

4.3.5.1 Introduction

This part is a discussion about and also a comparison of the resource needs of four object detection models. The models are *YOLOv8s*, *Faster R-CNN with MobileNetV3*, *SSD with VGG16*, and *EfficientDet D1*. The assessment is done with practical metrics that are essential

for real-world applications, such as the amount of GPU memory used, temperature increase, power consumption, CPU usage, RAM usage, and FPS. These metrics give us clues to the computational efficiency, energy requirements, and appropriateness of each model for real-time and low-resource scenarios.

4.3.5.2 Comparison

To assess the computational and resource efficiency of these models, the following metrics were evaluated:

- *GPU Memory Usage (MB)*: Indicates the amount of video memory (VRAM) utilized during inference, critical for evaluating memory-constrained environments.
- *GPU Temperature Increase (°C)*: Measures the rise in GPU temperature during inference, reflecting the thermal load on the hardware.
- *GPU Power Consumption (W)*: Represents the increase in power usage of the GPU during inference, essential for energy-efficient deployments.
- *CPU Usage (%)*: The percentage of CPU resources consumed during inference, highlighting the model's impact on the central processor.
- *RAM Usage (MB)*: The amount of system memory required for auxiliary tasks like data preprocessing and inference management.
- *FPS (Frames Per Second)*: Indicates the inference speed of the model, crucial for real-time performance evaluations.
- *Latency (ms)*: The time required for the model to process one image during inference.

Table 4.6: Comparison of models based on inference performance

Model	GPU Memory Usage (MB)	GPU Temp Increase (°C)	GPU Power (W)	CPU Usage (%)	RAM Usage (MB)	FPS	Latency (ms)
YOLOv8s	174.06	**4**	**59.83**	**0.6**	572.06	**482**	**2.08**
Faster R-CNN (MobileNetV3)	332.06	**4**	74.17	11.6	442.35	168	5.95
SSD (VGG16)	**136.00**	6	119.21	10.8	**346.43**	293	3.41
EfficientDet D1	172.06	5	69.61	6.5	473.57	96	10.42

4.3.5.3 Conclusions

The analysis of resource utilization and FPS points out some of the models' differences in their efficiency, computational requirements, and compatibility with real-time edge deployment as depicted in Table 4.6. The figures were obtained during the inference time with batch size being 1.

YOLOv8s is the most suitable model for real-time edge deployment as it combines computational efficiency with the best results in several hardware parameters. The power consumption of the GPU is the lowest with a value of just 59.83 W and CPU usage is extraordinarily low, 0.6%, thus indicating full GPU optimization and very little energy required from the system's central processor. Besides, YOLOv8s is the leader in the number of FPS with 482, and at the same time it has the lowest latency (2.08 ms), which guarantees a very quick reaction that is essential for instance in the case of wildfire detection. Although it uses a little bit more RAM (572 MB), this negative aspect is outweighed by its high performance, especially in real-time driven environments.

EfficientDet D1, which is designed in a more complex manner, compensates for the accuracy by the energy-efficiency ratio. It demonstrates medium values in most resource categories, such as GPU memory usage (172 MB), GPU power consumption (69.61 W), and latency (10.42 ms). Not being the fastest one, still, it can be used in cases where the balance between performance and resource constraints is required. Its losses of CPU at 6.5% and RAM at 473 MB make it possible to deploy it on mid-range edge devices with reasonable resource support.

Faster R-CNN (MobileNetV3), which is built with a light-weight feature extractor, still struggles due to its two-stage detection pipeline. This causes the highest GPU memory usage (332 MB) and CPU usage (11.6%) along with quite low inference speed (168 FPS) and latency (5.95 ms). Thus, these characteristics make it less suitable for real-time applications on a resource-limited platform, even though it has good detection accuracy.

SSD (VGG16) is the most energy-efficient GPU memory-wise (136 MB) and power-efficient RAM-wise (346 MB) in the test, which makes it suitable for devices with very limited resources. On the other hand, its high GPU power consumption (119.21 W) and considerable CPU usage (10.8%) indicate that the system is not running efficiently in power-limited environments. Additionally, although its FPS (293) and latency (3.41 ms) are good, they do not satisfy the requirements set by YOLOv8s for high-speed inference.

We can see in both Subsections 4.3.4 and 4.3.5 that the Yolov8s model has a suitable model complexity for deployment on edge devices as well as good real-time performance which is required in edge devices.

4.3.6 Training and Evaluation Time Comparison

4.3.6.1 Introduction

This section analyzes and compares the training and evaluation times of four object detection models: *YOLOv8s*, *Faster R-CNN with MobileNetV3*, *SSD with VGG16*, and *EfficientDet D1*. The comparison is rooted in the crucial timing indicators primarily focusing on the overall training time, the total number of epochs completed, the average training time per epoch, and the evaluation time needed to verify the models on separate RGB and thermal datasets. These indicators serve as a basis for each model's computational efficiency, training scalability, and response speed during inference evaluation.

4.3.6.2 Comparison

In order to evaluate the effectiveness of the detection models in their training and inference periods, we have documented and compared their training times, numbers of epochs, average time per epoch, and evaluation times. Evaluation time indicates how long each model will run on its corresponding test dataset. The results differ substantially among the architectures and datasets (RGB or thermal), especially if we take into account the large difference in the number of images between the datasets. Moreover, we utilized early stopping, which means that the training has been stopped as soon as convergence has been reached, thus explaining the different number of epochs for each model.

Table 4.7: Training and evaluation times for models on the RGB dataset

Model	Epochs	Training Time (h)	Training Time per Epoch (m)	Evaluation Time (s)
YOLOv8s	40	2.42	3.63	14.1
Faster R-CNN (MobileNetV3)	8	0.54	4.05	28.03
SSD (VGG16)	13	0.93	4.29	30.84
EfficientDet D1	13	2.02	9.32	46.3

Table 4.8: Training and evaluation times for models on the Thermal dataset

Model	Epochs	Training Time (h)	Training Time per Epoch (m)	Evaluation Time (s)
YOLOv8s	20	2.84	8.52	31.65
Faster R-CNN (MobileNetV3)	9	1.29	8.6	37.48
SSD (VGG16)	12	1.87	9.35	42.73
EfficientDet D1	14	5.32	22.8	66.77

4.3.6.3 Conclusions

Observing the training and evaluation metrics of models across Tables 4.7 and 4.8 offers a wealth of insight into the behavior of their performance characteristics. To cite some examples, it is evident that models such as SSD (VGG16) and Faster R-CNN (MobileNetV3) have a tendency to quickly overfit to the training data due primarily to their higher model complexity and larger number of parameters. It should be noted that Faster R-CNN with the MobileNetV3 backbone appears to be already exhibiting overfitting symptoms during the 2nd or 3rd epoch at the latest. This kind of premature learning of the training set is a phenomenon that occurs with complex models, particularly when dealing with complicated tasks or high-dimensional datasets.

The comparison of the training time per epoch shows that SSD and Faster R-CNN have longer training runs (4.29 minutes and 4.05 minutes, respectively, on the RGB dataset, and 9.35 and 8.6 minutes, respectively, on the thermal dataset). This is in accordance with their complex backbones and large numbers of parameters. These longer training times also imply that the evaluation times will be extended since the models that require a lot of computation will naturally take more time during inference.

On the other hand, EfficientDet D1, which has the fewest parameters, is no faster in terms of training or evaluation than YOLOv8s. This surprising finding is elucidated by the architectural complexity of EfficientDet. The employment of feature fusion and multi-scale processing in EfficientDet is such a high-tech feature that it increases the computational load. Hence, EfficientDet manifests large per-epoch training times (9.32 minutes for RGB and 22.8 minutes for thermal) as well as long evaluation intervals of up to 46.3 seconds and 66.77

seconds, correspondingly.

The significantly larger thermal dataset, that includes 97,422 images in comparison with only 38,993 in the RGB dataset, understandably results in increased training and evaluation times for all models. More data means more computations, hence longer training and evaluation periods are needed for the models to adjust and test.

Among all models, YOLOv8s consistently proves to be the most efficient. It has the lowest per-epoch training time (3.63 minutes for RGB and 8.52 minutes for thermal data) and the quickest evaluation time (14.1 seconds for RGB and 31.65 seconds for thermal). Its single-stage detection design and optimized architecture allow it to be both fast and overfitting resistant. These features make YOLOv8s ideal for real-time applications and running on edge devices.

4.3.7 Performance Analysis Based on Detection Metrics

4.3.7.1 Introduction

This section gives a very detailed performance analysis of object detection models that were tested, based on the most important detection metrics. It includes data from both RGB and thermal datasets and includes all four models: *YOLOv8s*, *Faster R-CNN with MobileNet-V3*, *SSD with VGG16*, and *EfficientDet D1*. The evaluation is organized in a way that allows mapping each model's behavior and accuracy trends across modalities and dataset-specific challenges.

Every model had a set of hyperparameters that were uniquely suited for the performance optimization, but at the same time, these hyperparameters made it possible to make a fair comparison across architectures. These hyperparameters, such as learning rate, batch size, optimizer type and many more, are documented and consistent throughout the evaluation, ensuring the validity and reproducibility of the results.

Loss curves during training and validating, as well as the changes of the key results like precision, recall, F1 score, mean Average Precision at different IoU thresholds (mAP@0.5, mAP@0.75), and the COCO-style metric mAP@0.5:0.95 are all part of the evidence. These numbers are displayed in charts to depict the improvement of a model's performance for training epochs, allowing the detection of convergence behavior and excessive fitting problems.

In addition to this, a comprehensive per-class evaluation is conducted where class-wise

and overall values for precision, recall, F1 score, and mAP are provided. This detailed investigation of each model allows a better understanding of how well the system fits to each class of objects (such as fire, smoke, human, animal), allowing a more targeted assessment of strengths and weaknesses per detection task.

The section concludes with a comparative discussion to identify the most suitable model for each dataset, considering both quantitative performance and class-specific effectiveness. This performance-based evaluation informs the selection of the best detection architecture under varying environmental and modality constraints.

4.3.7.2 Performance Analysis of YOLOv8s in the RGB Dataset

Table 4.9: YOLOv8s hyperparameters (RGB dataset)

Hyperparameter	Value
Batch size	8
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.01
Warm-up learning rate	0.1
Warm-up epochs	3
Weight decay	0.0005
Momentum	0.937
Patience (early stopping)	10
Maximum epochs	40
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.25

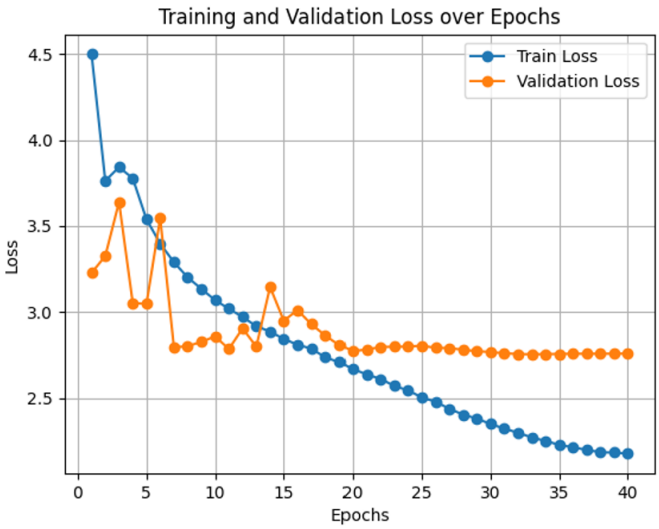


Figure 4.32: Training and validation loss curves for YOLOv8s (RGB dataset)

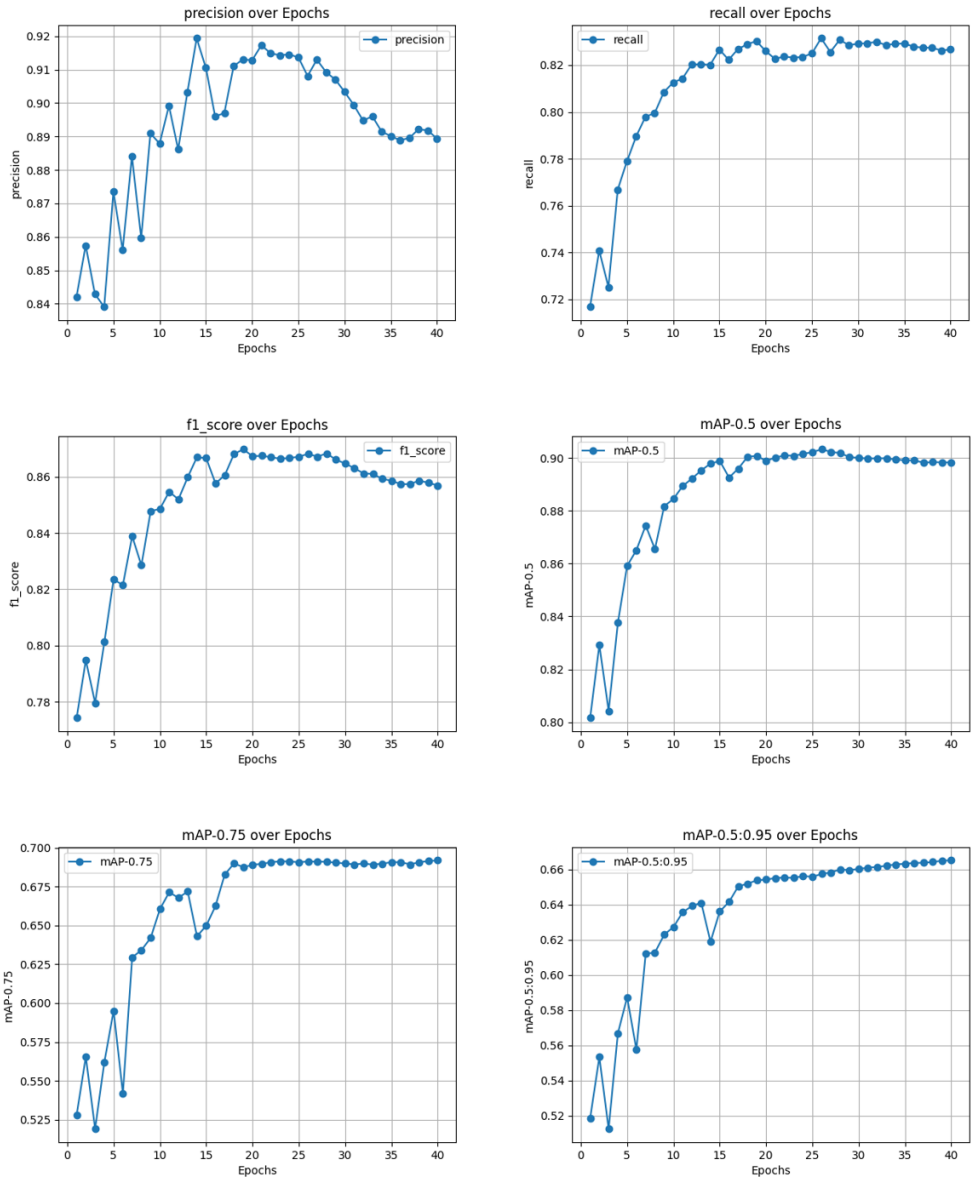


Figure 4.33: Plots per epoch for YOLOv8s (RGB dataset)

Observations on Figure 4.32

The *training loss* steadily decreases over the epochs, indicating effective learning and optimization throughout the training process.

The *validation loss* stabilizes after approximately epoch 20, suggesting that the model begins to generalize well without exhibiting signs of overfitting.

The relatively smooth convergence in both training and validation loss demonstrates the stability of YOLOv8s during training. This behavior aligns with the strong evaluation performance observed across the validation and test datasets, reinforcing the robustness and reliability of the model.

4.3.7.3 Performance Analysis of YOLOv8s in the Thermal Dataset

Table 4.10: YOLOv8s hyperparameters (Thermal dataset)

Hyperparameter	Value
Batch size	16
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.001
Warm-up learning rate	0.01
Warm-up epochs	3
Weight decay	0.001
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.25

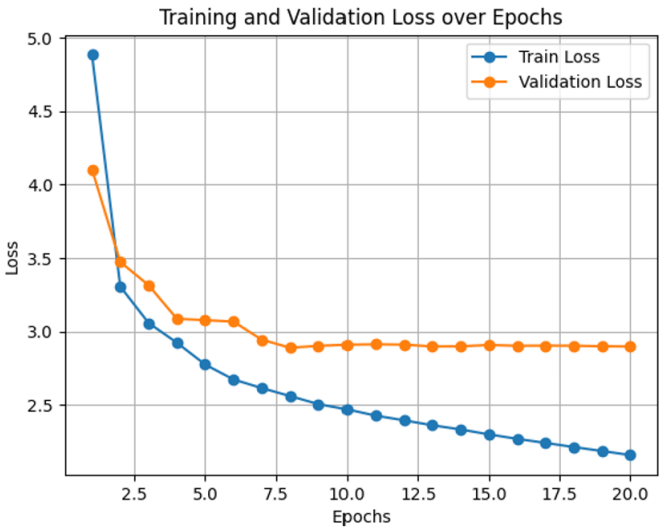


Figure 4.34: Training and validation loss curves for YOLOv8s (Thermal dataset)

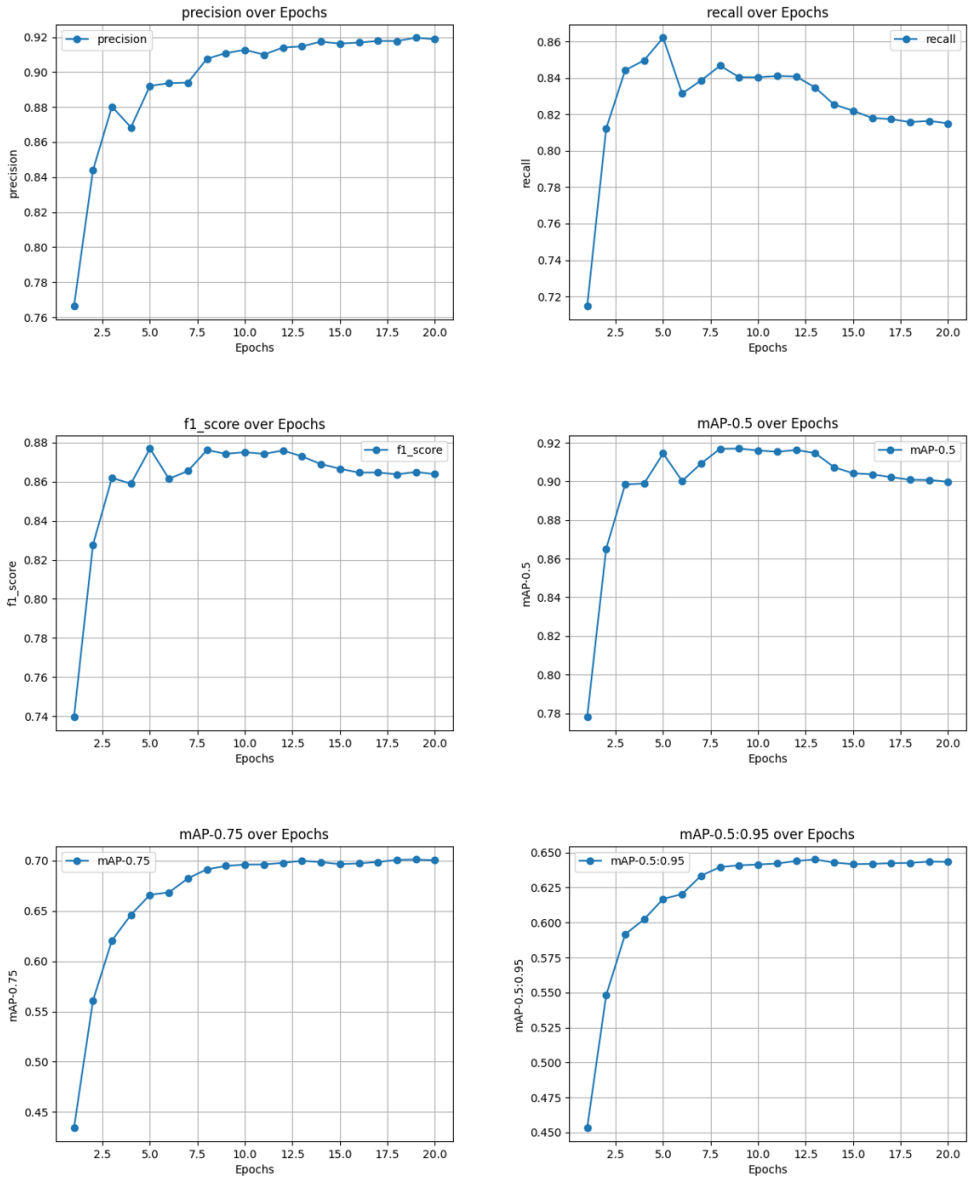


Figure 4.35: Plots per epoch for YOLOv8s (Thermal dataset)

Observations on Figure 4.34

The *training loss* steadily decreases over the epochs, indicating effective learning and optimization.

The *validation loss* stabilizes after epoch 8, suggesting that the model generalizes well without overfitting.

Finally, the key difference between the *RGB* and *Thermal* datasets (Figure 4.32 and Figure 4.34, respectively) is that in the Thermal dataset, the training process exhibits *smoother convergence* with fewer fluctuations at the beginning of training. This suggests that the model adapts more steadily to the thermal data, whereas in the RGB dataset, the initial fluctuations

indicate greater variability or complexity in learning the features from the dataset.

4.3.7.4 Performance Analysis of Faster R-CNN with MobileNetV3 in the RGB Dataset

Table 4.11: Faster R-CNN (MobileNetV3) hyperparameters (RGB dataset)

Hyperparameter	Value
Batch size	8
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.0001
Warm-up learning rate	0.00001
Warm-up epochs	3
Weight decay	0.001
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.45

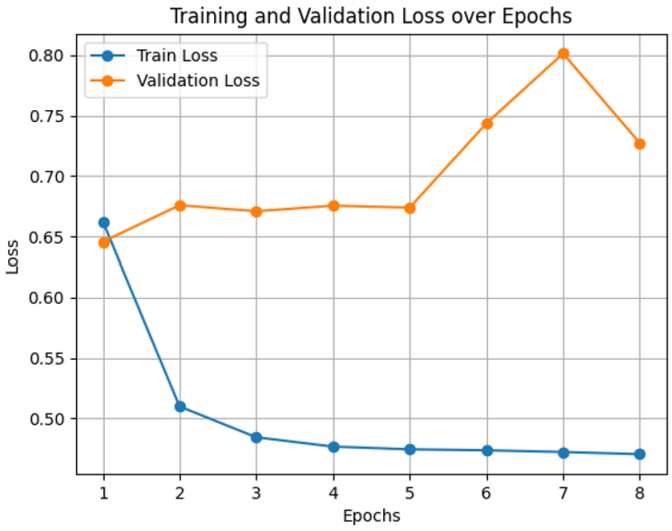


Figure 4.36: Training and validation loss curves for Faster R-CNN with MobileNetV3 (RGB dataset)

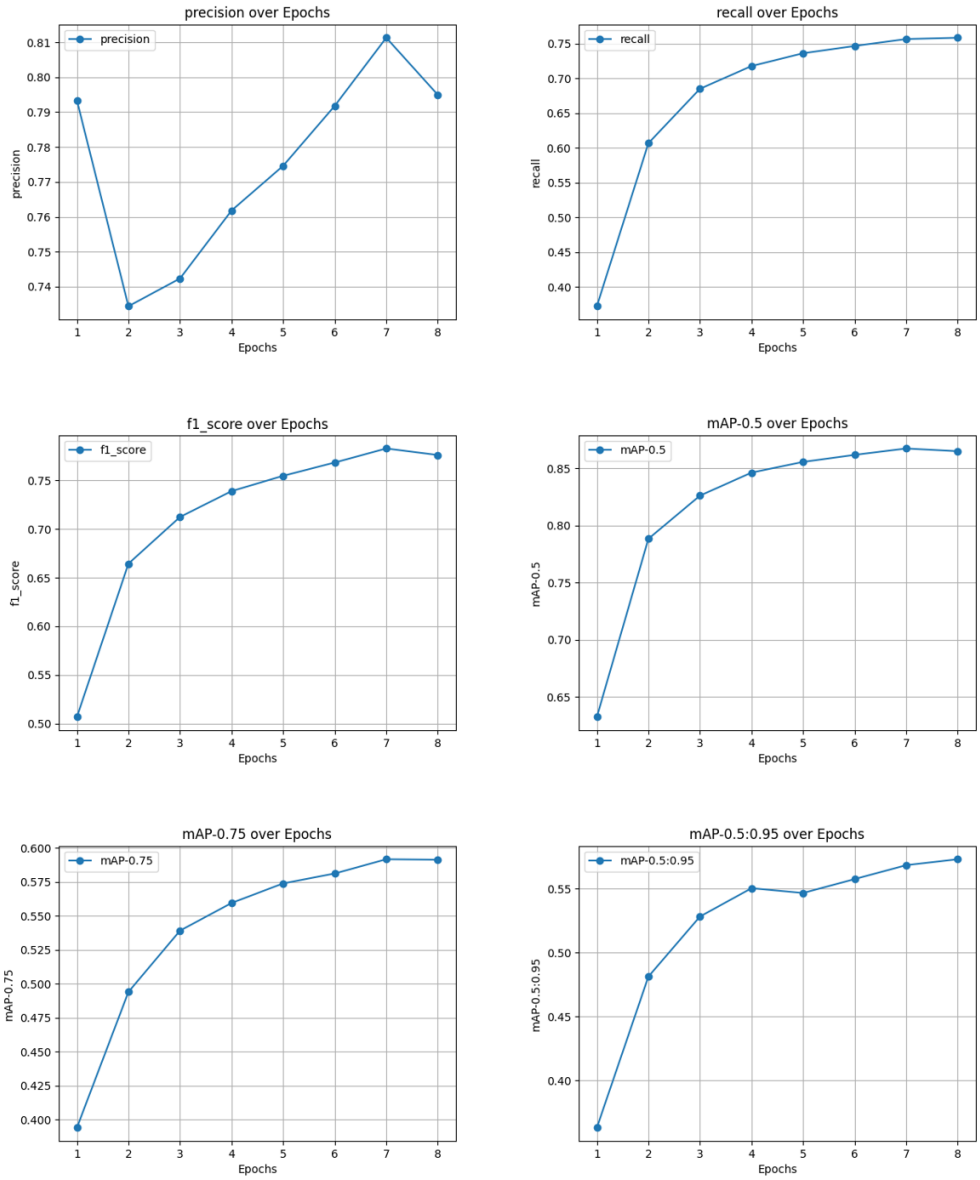


Figure 4.37: Plots per epoch for Faster R-CNN with MobileNetV3 (RGB dataset)

Observations on Figure 4.36

The *training loss* decreases steadily, but the *validation loss* fluctuates significantly, particularly after epoch 5.

This divergence suggests potential *overfitting* or sensitivity to noisy validation data. Despite the early stopping strategy, the fluctuations in validation loss imply suboptimal generalization.

4.3.7.5 Performance Analysis of Faster R-CNN with MobileNetV3 in the Thermal Dataset

Table 4.12: Faster R-CNN (MobileNetV3) hyperparameters (Thermal dataset)

Hyperparameter	Value
Batch size	16
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.0001
Warm-up learning rate	0.00001
Warm-up epochs	3
Weight decay	0.001
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.35

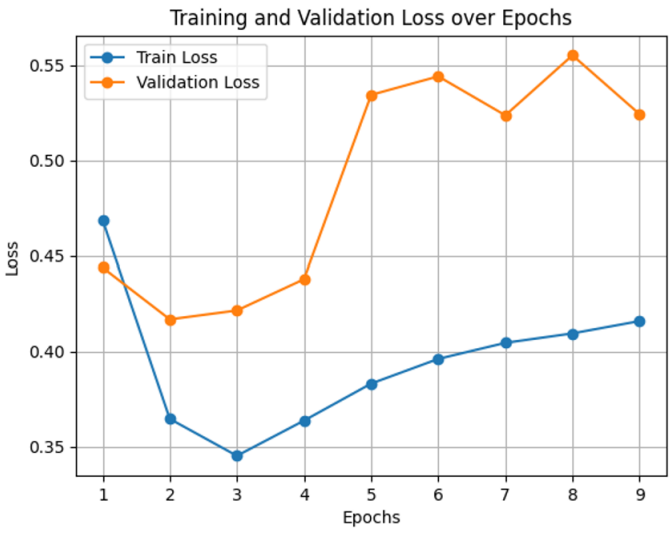


Figure 4.38: Training and validation loss curves for Faster R-CNN with MobileNetV3 (Thermal dataset)

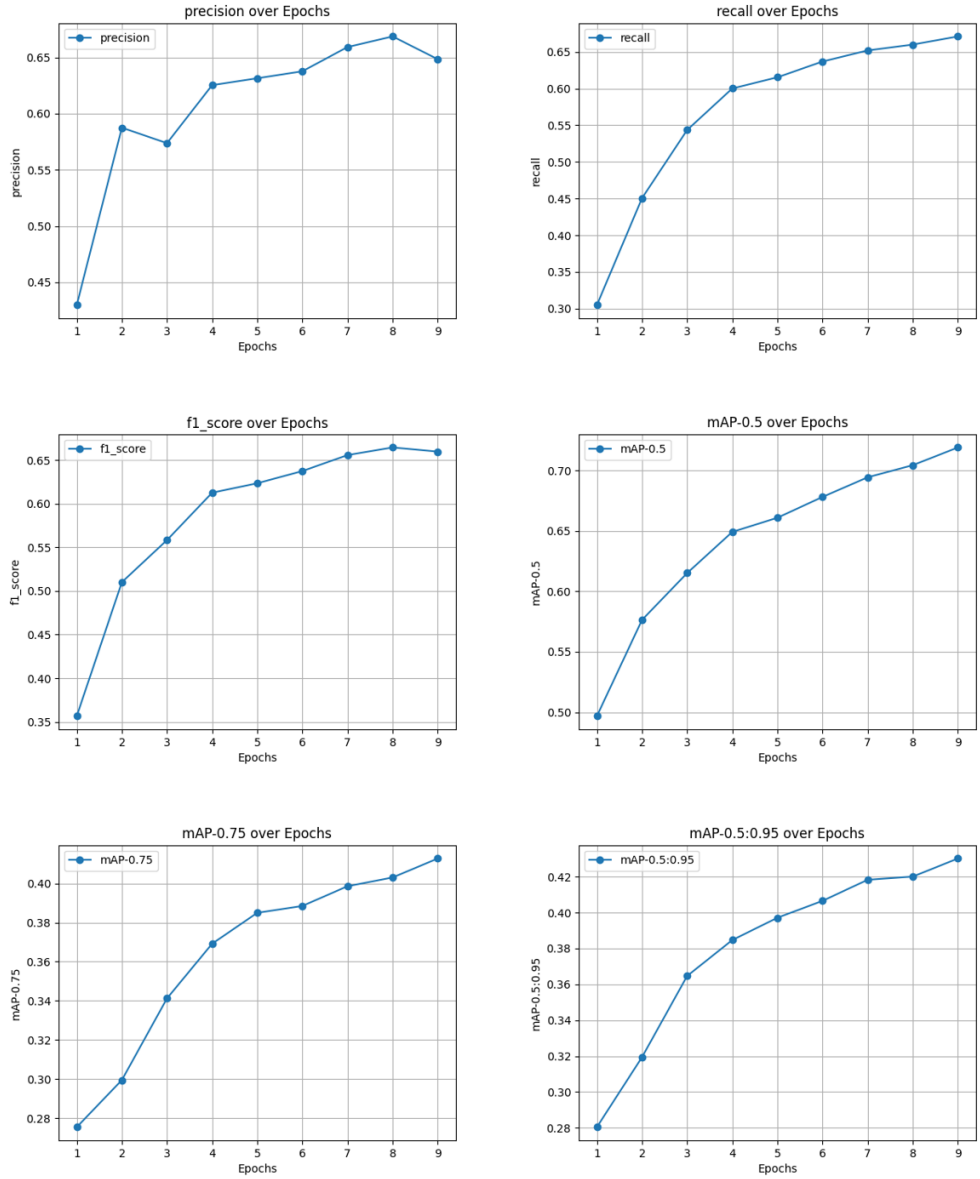


Figure 4.39: Plots per epoch for Faster R-CNN with MobileNetV3 (Thermal dataset)

Observations on Figure 4.38

The *training loss* decreases steadily, but in epoch 3 it starts to increase.

The *validation loss* increases after epoch 2 and fluctuates significantly, particularly after epoch 5.

Like in the RGB dataset (Figure 4.36), this divergence again suggests potential *overfitting* or sensitivity to noisy validation data, indicating a suboptimal generalization.

4.3.7.6 Performance Analysis of SSD with VGG16 in the RGB Dataset

Table 4.13: SSD (VGG16) hyperparameters (RGB dataset)

Hyperparameter	Value
Batch size	8
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.0001
Warm-up learning rate	0.00001
Warm-up epochs	3
Weight decay	0.001
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.25

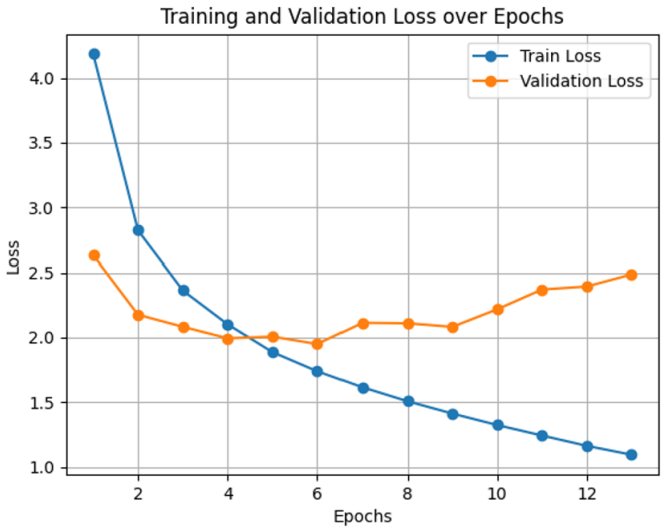


Figure 4.40: Training and validation loss curves for SSD with VGG16 (RGB dataset)

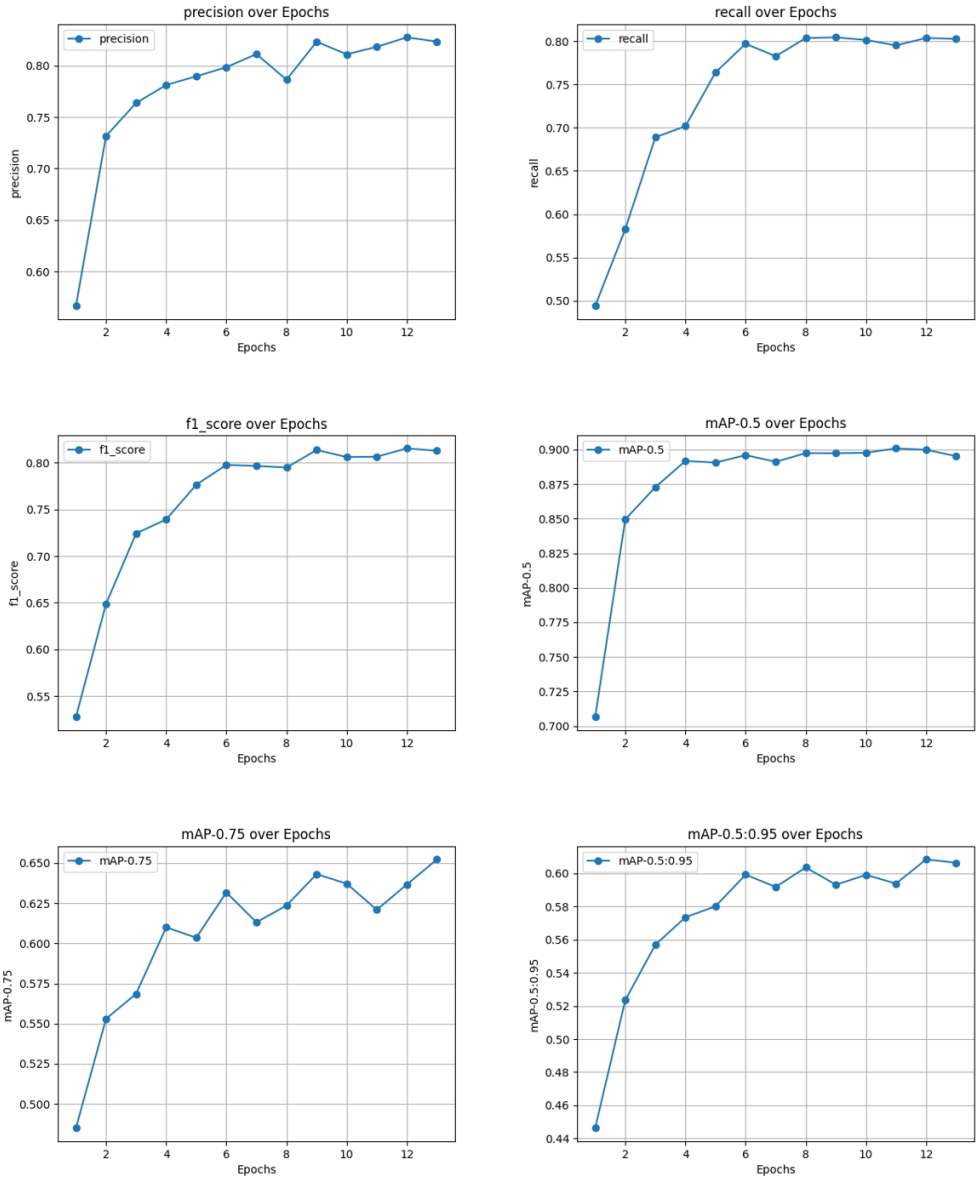


Figure 4.41: Plots per epoch for SSD with VGG16 (RGB dataset)

Observations on Figure 4.40

The *training loss* decreases steadily, indicating that the model learns effectively during the training phase.

The *validation loss* stabilizes early around epoch 6 but starts to increase slightly in the later epochs. This suggests the onset of *overfitting*, where the model performs well on the training data but starts to lose its ability to generalize to unseen data.

The gap between training and validation losses widens as training progresses, further highlighting overfitting. The validation loss doesn't sharply increase, but the upward trend is noticeable.

4.3.7.7 Performance Analysis of SSD with VGG16 in the Thermal Dataset

Table 4.14: SSD (VGG16) hyperparameters (Thermal dataset)

Hyperparameter	Value
Batch size	16
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.0001
Warm-up learning rate	0.00001
Warm-up epochs	3
Weight decay	0.001
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.25

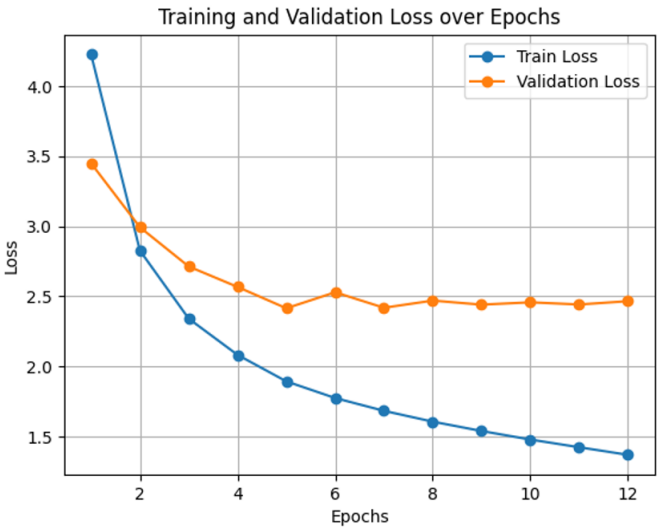


Figure 4.42: Training and validation loss curves for SSD with VGG16 (Thermal dataset)

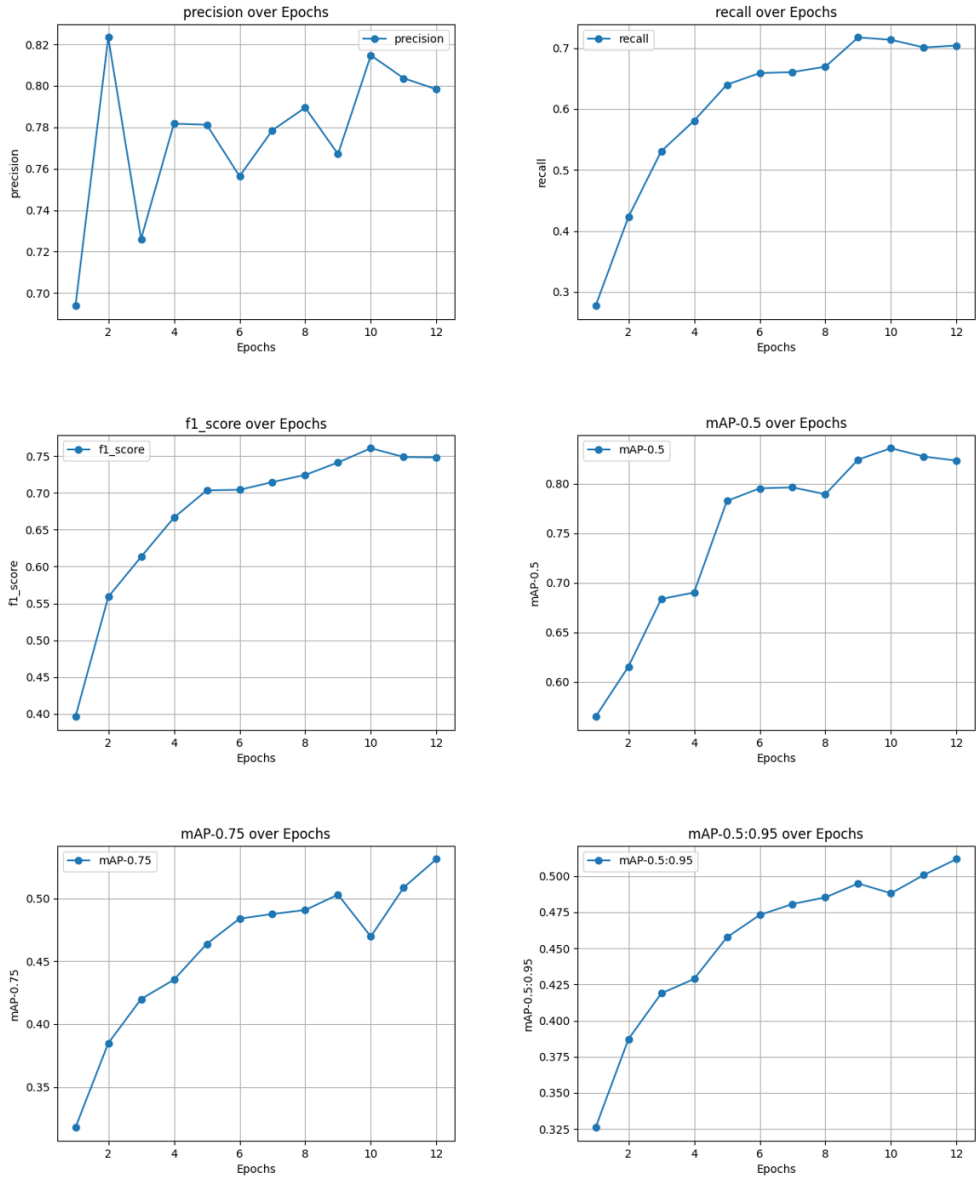


Figure 4.43: Plots per epoch for SSD with VGG16 (Thermal dataset)

Observations on Figure 4.42

Similar to previous observations(Figure 4.43), the *training loss* in the Thermal dataset decreases steadily, while the *validation loss* stabilizes after epoch 5 without significant fluctuations.

In contrast, in the RGB dataset, the validation loss not only fluctuates but also shows an increasing trend at certain points, indicating potential overfitting or greater variability in the data.

This suggests that the Thermal dataset is simpler than the RGB dataset, allowing for smoother convergence and more stable training dynamics.

4.3.7.8 Performance Analysis of EfficientDet D1 in the RGB Dataset

Table 4.15: EfficientDet D1 hyperparameters (RGB dataset)

Hyperparameter	Value
Batch size	8
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.001
Warm-up learning rate	0.0001
Warm-up epochs	3
Weight decay	0.0005
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.3

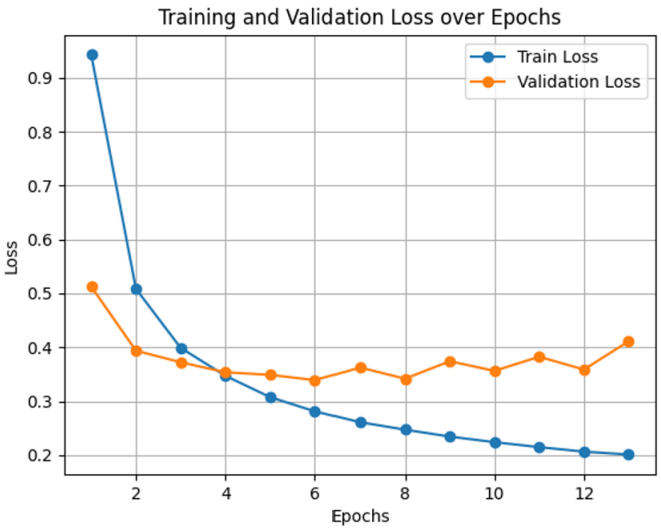


Figure 4.44: Training and validation loss curves for EfficientDet D1 (RGB dataset)

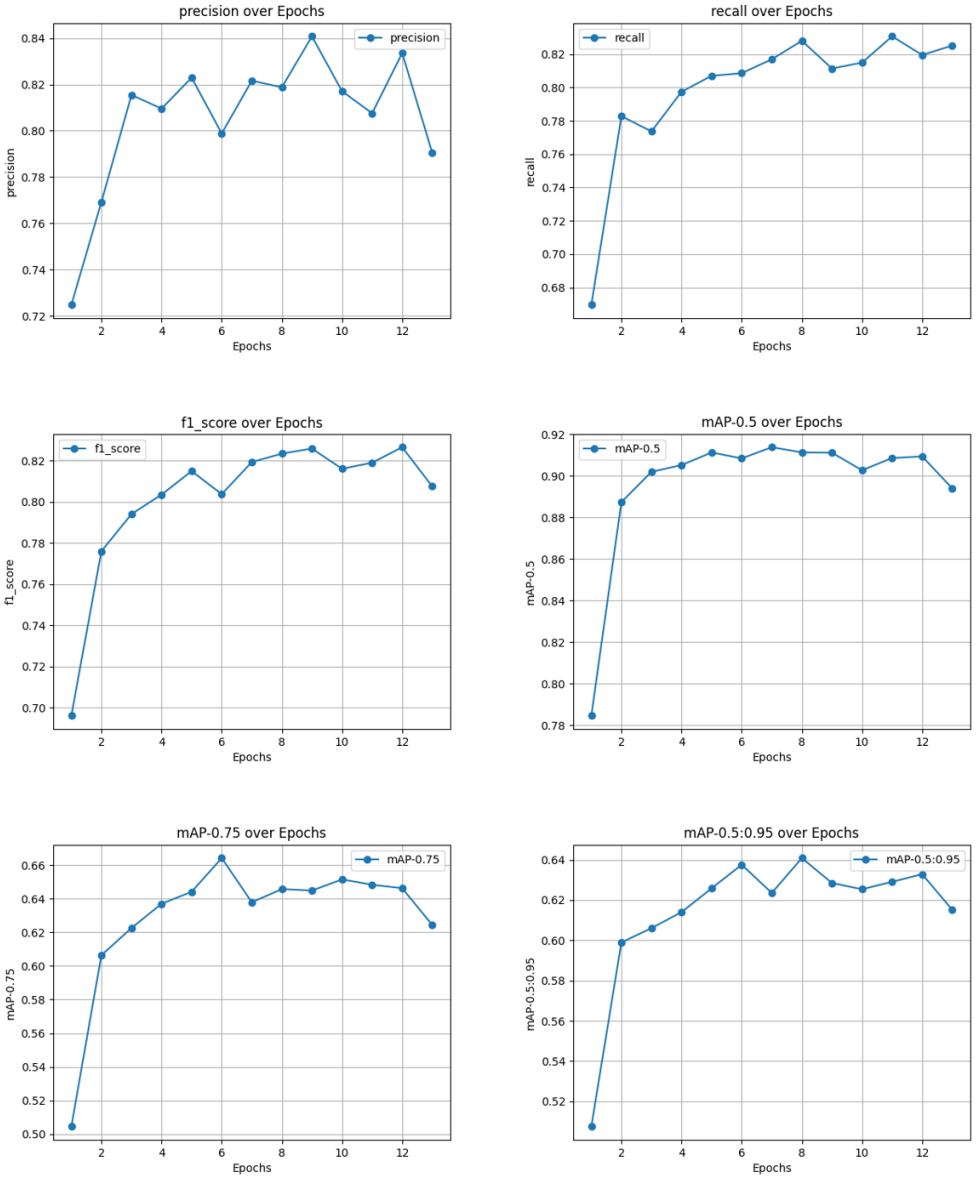


Figure 4.45: Plots per epoch for EfficientDet D1 (RGB dataset)

Observations on Figure 4.44

The *training loss* decreases steadily and smoothly, reflecting effective learning and optimization during the training process.

The *validation loss* stabilizes early (around epoch 4) but begins to exhibit small fluctuations toward the later epochs. This suggests a minor degree of *overfitting*, though it is not severe enough to heavily impact the model's performance.

The gap between the training and validation losses remains consistent, indicating that the model maintains a reasonable balance between learning and generalization.

4.3.7.9 Performance Analysis of EfficientDet D1 in the Thermal Dataset

Table 4.16: EfficientDet D1 hyperparameters (Thermal dataset)

Hyperparameter	Value
Batch size	16
Optimizer	SGD
Learning rate scheduler	cosine
Learning rate	0.001
Warm-up learning rate	0.0001
Warm-up epochs	3
Weight decay	0.0005
Momentum	0.937
Patience (early stopping)	7
Maximum epochs	30
Number of workers	16
Confidence threshold for balanced val/test precision and recall	0.3

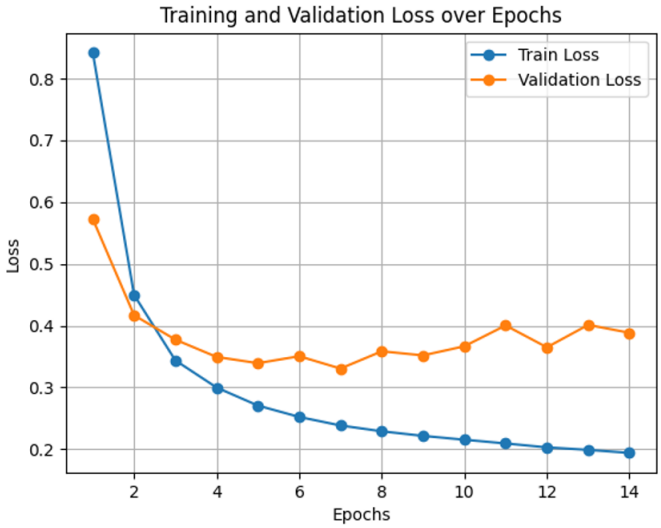


Figure 4.46: Training and validation loss curves for EfficientDet D1 (Thermal dataset)

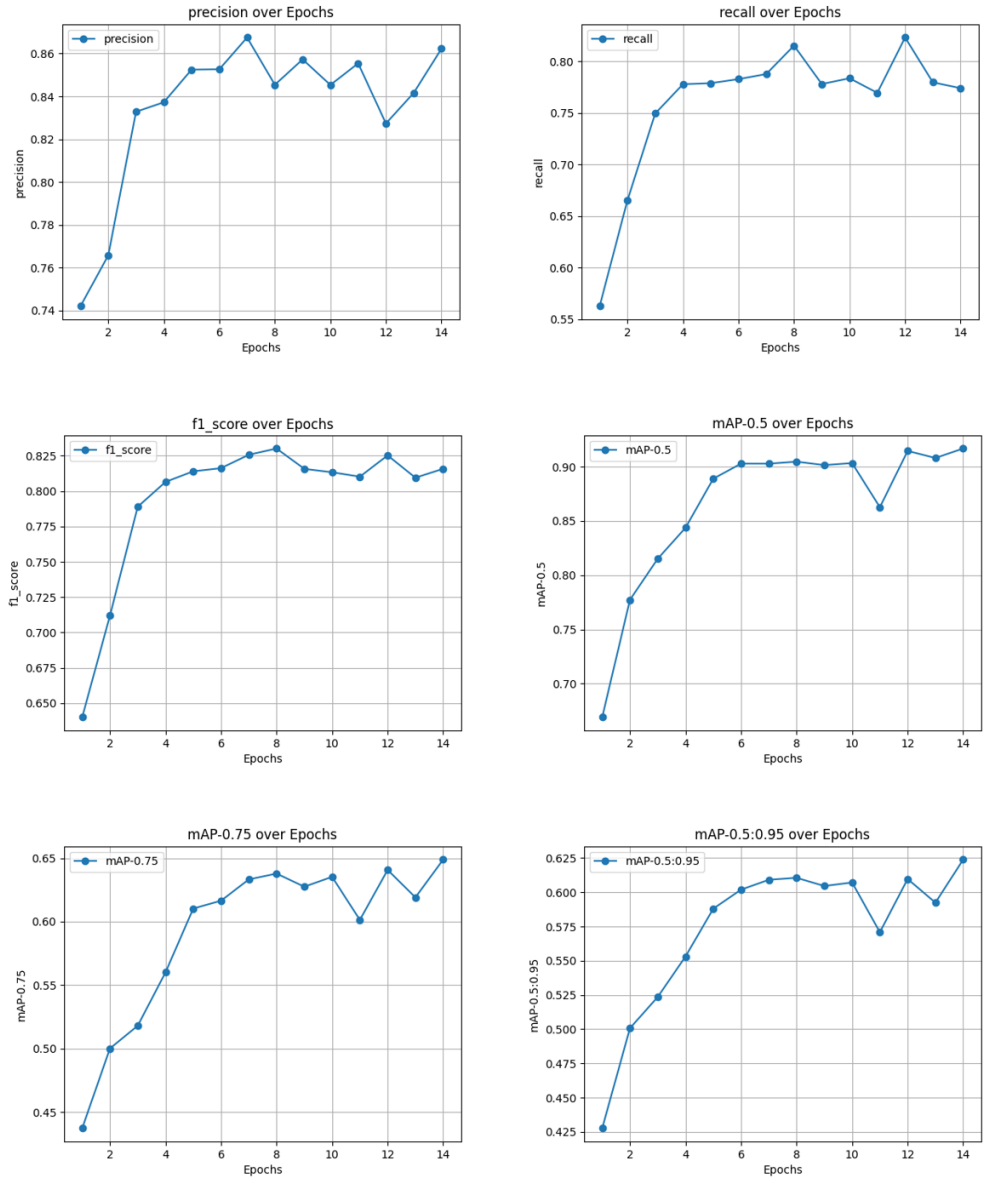


Figure 4.47: Plots per epoch for EfficientDet D1 (Thermal dataset)

Observations on Figure 4.46

The *training loss* decreases steadily, but the *validation loss* exhibits some fluctuations after epoch 6, indicating a slight overfitting to the training data. However, compared to the RGB dataset (Figure 4.44), these fluctuations are less pronounced, supporting previous observations about SSD and YOLOv8s. This reinforces the idea that the Thermal dataset allows for more stable convergence, whereas the RGB dataset presents greater variability during training.

This is logical because the Thermal dataset, particularly for human and animal detection, has fewer variations in object sizes compared to the RGB dataset, where fire and smoke ex-

hibit a wider range of shapes and scales. This difference in object size variability contributes to the smoother convergence observed in the Thermal dataset, while the RGB dataset experiences more fluctuations due to the complexity and diversity of object appearances.

4.3.7.10 Per-Class Evaluation Tables for the RGB Dataset

Table 4.17: Overall class performance metrics on the RGB dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.890**	0.826	**0.857**	0.898	**0.692**	**0.665**
YOLOv8s	Test	**0.903**	**0.849**	**0.875**	**0.913**	**0.664**	**0.632**
Faster R-CNN (MobileNetV3)	Val	0.795	0.758	0.776	0.865	0.591	0.573
Faster R-CNN (MobileNetV3)	Test	0.796	0.729	0.761	0.856	0.486	0.513
SSD (VGG16)	Val	0.827	0.804	0.815	0.900	0.637	0.608
SSD (VGG16)	Test	0.832	0.784	0.808	0.896	0.584	0.560
EfficientDet D1	Val	0.819	**0.828**	0.823	**0.911**	0.646	0.641
EfficientDet D1	Test	0.818	0.805	0.811	0.907	0.611	0.588

Table 4.18: Fire class performance metrics on the RGB dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.830**	**0.739**	**0.782**	0.833	**0.492**	**0.503**
YOLOv8s	Test	**0.898**	**0.803**	**0.848**	**0.885**	**0.596**	**0.558**
Faster R-CNN (MobileNetV3)	Val	0.651	0.626	0.638	0.776	0.391	0.428
Faster R-CNN (MobileNetV3)	Test	0.700	0.646	0.672	0.800	0.435	0.441
SSD (VGG16)	Val	0.732	0.681	0.706	0.828	0.438	0.472
SSD (VGG16)	Test	0.791	0.705	0.745	0.855	0.496	0.494
EfficientDet D1	Val	0.708	0.716	0.712	**0.847**	0.461	0.494
EfficientDet D1	Test	0.769	0.716	0.742	0.870	0.465	0.502

Table 4.19: Smoke class performance metrics on the RGB dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.950**	0.913	0.931	0.964	**0.890**	**0.827**
YOLOv8s	Test	**0.908**	**0.895**	**0.902**	0.942	0.731	**0.705**
Faster R-CNN (MobileNetV3)	Val	0.939	0.891	0.914	0.955	0.792	0.719
Faster R-CNN (MobileNetV3)	Test	0.893	0.813	0.851	0.912	0.537	0.584
SSD (VGG16)	Val	0.923	0.926	0.925	0.972	0.835	0.744
SSD (VGG16)	Test	0.874	0.864	0.869	0.936	0.672	0.625
EfficientDet D1	Val	0.930	**0.940**	**0.935**	**0.976**	0.831	0.787
EfficientDet D1	Test	0.866	0.894	0.880	**0.945**	**0.756**	0.674

4.3.7.11 Per-Class Evaluation Tables for the Thermal Dataset

Table 4.20: Overall class performance metrics on the Thermal dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.915**	**0.834**	**0.873**	0.915	**0.700**	**0.645**
YOLOv8s	Test	**0.886**	**0.813**	**0.848**	**0.878**	**0.589**	**0.556**
Faster R-CNN (MobileNetV3)	Val	0.648	0.671	0.660	0.719	0.413	0.430
Faster R-CNN (MobileNetV3)	Test	0.686	0.670	0.678	0.694	0.402	0.406
SSD (VGG16)	Val	0.798	0.704	0.748	0.823	0.531	0.512
SSD (VGG16)	Test	0.803	0.678	0.736	0.750	0.419	0.428
EfficientDet D1	Val	0.862	0.774	0.816	**0.917**	0.649	0.624
EfficientDet D1	Test	0.819	0.753	0.784	0.818	0.462	0.471

Table 4.21: Fire class performance metrics on the Thermal dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	0.885	0.865	0.875	0.922	0.832	0.781
YOLOv8s	Test	**0.940**	0.829	0.881	0.917	**0.751**	0.655
Faster R-CNN (MobileNetV3)	Val	0.821	**0.948**	**0.880**	0.975	0.749	0.717
Faster R-CNN (MobileNetV3)	Test	0.910	**0.960**	**0.935**	**0.981**	0.718	**0.668**
SSD (VGG16)	Val	0.839	0.785	0.811	0.846	0.747	0.645
SSD (VGG16)	Test	0.922	0.838	0.878	0.910	0.564	0.549
EfficientDet D1	Val	**0.899**	0.842	0.870	**0.991**	**0.926**	**0.837**
EfficientDet D1	Test	0.900	0.903	0.901	0.945	0.568	0.562

Table 4.22: Human class performance metrics on the Thermal dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.953**	**0.897**	**0.924**	**0.952**	**0.715**	**0.634**
YOLOv8s	Test	**0.918**	**0.855**	**0.885**	**0.905**	0.519	0.533
Faster R-CNN (MobileNetV3)	Val	0.686	0.709	0.697	0.759	0.361	0.401
Faster R-CNN (MobileNetV3)	Test	0.735	0.720	0.727	0.736	0.375	0.397
SSD (VGG16)	Val	0.911	0.765	0.832	0.887	0.513	0.518
SSD (VGG16)	Test	0.910	0.744	0.819	0.831	0.514	0.502
EfficientDet D1	Val	0.896	0.809	0.850	0.908	0.560	0.560
EfficientDet D1	Test	0.862	0.777	0.817	0.870	**0.654**	**0.540**

Table 4.23: Animal class performance metrics on the Thermal dataset

Model	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
YOLOv8s	Val	**0.907**	**0.740**	**0.815**	**0.869**	**0.553**	**0.521**
YOLOv8s	Test	**0.800**	**0.754**	**0.776**	**0.813**	**0.496**	**0.479**
Faster R-CNN (MobileNetV3)	Val	0.438	0.356	0.393	0.423	0.128	0.173
Faster R-CNN (MobileNetV3)	Test	0.412	0.330	0.366	0.364	0.111	0.154
SSD (VGG16)	Val	0.645	0.562	0.600	0.736	0.334	0.372
SSD (VGG16)	Test	0.578	0.453	0.508	0.509	0.178	0.234
EfficientDet D1	Val	0.792	0.671	0.726	0.852	0.464	0.475
EfficientDet D1	Test	0.695	0.577	0.631	0.639	0.269	0.312

4.3.7.12 Conclusions and Best Model Selection for RGB Dataset

This section concludes the evaluation of four object detection models (*YOLOv8s*, *EfficientDet D1*, *SSD with VGG16*, and *Faster R-CNN*) on the RGB dataset with *fire* and *smoke* classes. Using the metrics provided in the previous sections, including precision, recall, F1-score, and mAP (at IoU thresholds of 0.5, 0.75, and 0.5:0.95) across both validation and test datasets, the most effective model for this task is identified. The analysis aims to determine the model that offers the best combination of accuracy, reliability, and generalization for detecting fire and smoke.

Fire Detection

YOLOv8s is the most trusted model for fire detection. It sets a record by achieving the highest precision, recall, and F1-scores in the validation and test datasets. Besides, YOLOv8s always is the leader in mAP values at stricter thresholds (mAP@0.75 and mAP@0.5:0.95), showing good generalization and stability on unseen data. That means YOLOv8s is the most reliable and versatile model for fire detection tasks.

Faster R-CNN with MobileNetV3 is at the lowest end of the spectrum in fire detection metrics and trails behind other models by a wide margin. The lack of generalization in the fire detection task of the model is quite apparent in the lower precision, recall, and mAP values. This is visible also in the validation loss plot, which portrays the model's inability to train steadily due to its fluctuating losses, thus resulting in poor performance.

SSD with VGG16 is a good choice for fire detection, as it fairly matches the precision, recall, and F1-scores of other models. Yet, it is not that good as YOLOv8s and EfficientDet D1 at stricter IoU thresholds, which we can see from the lower mAP@0.75 and mAP@0.5:0.95 scores. This means that there is a compromise between precision and recall in more difficult conditions, and this makes it less effective for fire detection than the top-performing models.

EfficientDet D1 informs as well about solid performance in fire detection in terms of precision, recall, and F1-scores. Though it scores well on validation data and surpasses some models, it falls behind YOLOv8s at stricter thresholds, especially on the test dataset. This implies that while EfficientDet D1 is still a good solution, YOLOv8s is the better option for fire detection, particularly in various and unknown environments.

Smoke Detection

YOLOv8s offers a strong and balanced performance for smoke detection, delivering high precision, recall, and F1-scores across all metrics. Although it is slightly behind EfficientDet D1 in mAP@0.5 for smoke detection, YOLOv8s still maintains a better overall balance if fire detection is also considered. Its generalization to unseen data further underscores its reliability.

Faster R-CNN with MobileNetV3 is a little way off in performance expectancy in smoke detection. It is recall wise very resilient and mAP@0.5 good. It has, however, consistent problems on stricter IoU thresholds, where it falls short of YOLOv8s and EfficientDet D1, besides that it is not good in generalization across validation and test datasets. Therefore, the situation is not very favorable for it if precision and stability are needed.

SSD with VGG16 performs competitively for smoke detection, providing a reasonable precision and F1-score. Still, at more rigorous thresholds (mAP@0.75 and mAP@ 0.5:0.95), its performance decreases and becomes inferior to the two: YOLOv8s and EfficientDet D1. This means it cannot be applied with full confidence where higher precision is demanded.

EfficientDet D1 is the best smoke detector, with the highest mAP@0.5 numbers on both validation and test data sets. Furthermore, it offers considerable recall and F1 scores; thus, the model can be particularly good at finding smoke instances and doing it correctly and consistently. However, it is a little bit behind YOLOv8s in the case of stricter thresholds (mAP@0.75 and mAP@0.5:0.95), which represents a slight decrease in precision as the IoU condition is higher.

Overall Performance

YOLOv8s is the best-performing model overall, achieving the highest precision, recall, and F1-scores for both fire and smoke detection. It balances performance across all metrics while demonstrating strong generalization from validation to test datasets. Its exceptional performance in both fire and smoke detection make it the most versatile and reliable model.

Faster R-CNN with MobileNetV3 is the weakest model overall, particularly for fire detection, where it struggles significantly. While it shows potential for smoke detection, it lacks consistency and generalization across datasets, making it the least reliable option.

SSD with VGG16 delivers moderate performance overall. While it is reliable for smoke detection, it is less effective for fire detection compared to YOLOv8s and EfficientDet D1. Its weaker mAP values at stricter thresholds further highlight its limitations.

EfficientDet D1 ranks as a close second, excelling particularly in smoke detection while maintaining competitive performance in fire detection. It generalizes well and is a strong alternative, particularly for tasks prioritizing smoke detection accuracy.

Optimal Model Selection

The detailed analysis clearly establishes *YOLOv8s as the best-performing model, making it the optimal choice for the RGB dataset* by offering robust and balanced detection of both fire and smoke across all metrics. *EfficientDet D1* is a strong alternative for scenarios prioritizing smoke detection, while *SSD with VGG16* and *Faster R-CNN with MobileNetV3* provide functional but less consistent results. This comparison highlights *YOLOv8s* as the most versatile and reliable model for fire and smoke detection tasks.

4.3.7.13 Conclusions and Best Model Selection for Thermal Dataset

This section concludes the evaluation of four object detection models (*YOLOv8s*, *EfficientDet D1*, *SSD with VGG16*, and *Faster R-CNN with MobileNetV3*) on the thermal dataset with classes *fire*, *human*, and *animal*. Using the metrics provided, including precision, recall, F1-score, and mAP (at IoU thresholds of 0.5, 0.75, and 0.5:0.95) across validation and test datasets, the most effective model for this task is identified. The analysis evaluates each model's accuracy, reliability, and generalization to provide insights into their performance.

The metrics on the validation dataset are better than those on the test dataset, likely because the validation data is more similar to the training data compared to the test data. This similarity could lead to better performance on the validation set, as the model is better able

to generalize to data it has seen patterns of during training.

Fire Detection

YOLOv8s demonstrates the strongest performance for fire detection, achieving the highest precision, recall, and F1-scores across both validation and test datasets. It not only dominates mAP figures across various IoU thresholds (mAP@0.5, mAP@0.75, and mAP@0.5:0.95) but also gives consistent and reliable detection performance. YOLOv8s is undoubtedly the most trustworthy fire detection model with its great capacity for generalization on unseen test data, offering minimal trade-offs in real-world applications.

Faster R-CNN with MobileNetV3 achieves very high recall for fire detection but faces difficulties in precision and consistency in both validation and test datasets. Although it keeps comparable mAP figures between the validation and test datasets which shows that it has a good capacity for generalization even if the test data is different from the training set. Summing up, Faster R-CNN is ahead of SSD with VGG16 and EfficientDet D1 in many fire detection metrics, but it still can't defeat YOLOv8s in the overall accuracy and stability.

SSD with VGG16 gives average fire detection results but in most key metrics, it performs lower than YOLOv8s, EfficientDet D1, and Faster R-CNN with MobileNetV3. It obtains good precision and recall but it is unable to deliver the performance needed under more restrictive IoU thresholds (mAP@0.75 and mAP@0.5:0.95) thus it limits its application in the very precise fire detection cases. In the experiment, the model reveals weaker generalization to the test data which makes it less appropriate for solving fire detection tasks involving unseen variations problem.

EfficientDet D1 is the best fire detection model on the validation dataset, outperforming all other models, including YOLOv8s, in mAP@0.5 and stricter thresholds. Nevertheless, on the test dataset, YOLOv8s outplays EfficientDet D1 at higher IoU thresholds (mAP@0.75 and mAP@0.5:0.95), implying that YOLOv8s is able to generalize better to different test data that is not seen. In the case of EfficientDet D1, it has shown outstanding performance on the validation data, which means that it is well-suited to the training set, while YOLOv8s turns out to be more stable and flexible in real detection situations.

Human Detection

YOLOv8s represents a well-balanced human detection capability, precision, recall, and F1-scores that are very high in validation and test datasets. It outmatches other models even at stricter IoU thresholds such as mAP@0.75 and mAP@0.5:0.95, showing to be able to detect

humans with higher precision. However, there is a significant difference between the validation and test results, which could be explained by the validation set being more reflective of the training data than the test set, a pattern seen in other classes as well. Nevertheless, YOLOv8s still exhibits good generalization and robust human detection performance across the board.

Faster R-CNN with MobileNetV3 marks the lowest point on the performance scale for human detection, as it illustrates very poor precision, recall, and mAP values over all the metrics. The model definitely has a hard time generalizing efficiently hence it is inconsistent between the validation and test datasets. Due to the inability to recognize smaller items such as humans and animals, it most probably is less suitable for tasks that require that it carries out a fine detection of small entities, which is reasonable considering the limitations of its architecture in managing small-scale objects.

SSD with VGG16 can be considered as a moderate performer in human detection, as it reaches acceptable precision and recall. However, its F1-scores are minor in comparison to the scores of the YOLOv8s and EfficientDet D1, and at more rigorous IoU thresholds (mAP@0.75 and mAP@0.5:0.95) it experiences a sharp decrease in performance. The finding implies that SSD with VGG16 may have difficulties in detecting human subjects with precision. Still, its statistics between the validation and test sets are more constant than those of YOLOv8s, which means that it might generalize more consistently, although at a lower accuracy level.

EfficientDet D1 is the best one for human detection. It has very high precision, recall, and F1-scores for the validation and test data. On the other hand, it is a little bit weaker than YOLOv8s at the stricter IoU thresholds, which means that YOLOv8s might be more suitable for the tasks of human detection with very high precision. Still, EfficientDet D1 keeps the performance more stable across the validation and test datasets, thus showing better consistency when it is used with the unseen data.

Animal Detection

YOLOv8s provides the most robust performance for animal detection, achieving the highest precision, recall, and F1-scores across both validation and test datasets. It is by far the best performing model in mAP metrics across many datasets, particularly at stricter thresholds such as mAP@0.75 and mAP@0.5:0.95, which confirms that it is the most precise and reliable for animal detection. The model's high generalization capacity makes it the most

reliable one for animal instance detection, even when faced with unseen test data.

Faster R-CNN with MobileNetV3 demonstrates the weakest performance for animal detection, delivering the lowest precision, recall, F1-scores, and mAP values among all. The model cannot generalize well over validation and test datasets; therefore, it is untrustworthy for detecting animal instances. The situation is similar to that of human detection, where Faster R-CNN has a hard time detecting small objects correctly, which is still the main reason why its results for animals are poor as they are usually smaller in size compared to fire and human instances.

SSD with VGG16 shows average performance in animal detection, yielding lower mAP and F1-scores than YOLOv8s and EfficientDet D1. At stricter thresholds, its performance drops drastically as it is less able to generalize and is less reliable for precise animal detection. This may be because the animal instances in the dataset are smaller than other two classes, making them harder to detect accurately. What's more, the gap that exists between the validation and test dataset metrics shows that the validation data is probably more similar to the training data, whereas more variations in the test dataset cause performance to deteriorate.

EfficientDet D1 is an animal detector that is highly accurate at detecting objects at a threshold of 0.5 and decent at the F1 score, but it loses a small margin to YOLOv8s at more strict thresholds (mAP@0.75 and mAP@0.5:0.95). The model is quite efficient in using the training data to make reasonable assumptions about the test data, but the lower performance on the test set than on the validation set indicates that it might have more trouble with unseen data that are not similar to the training distribution. Similarly, SSD with VGG16, the difference between the results on validation and test sets shows that EfficientDet D1 is not that reliable on test data, thus reinforcing that YOLOv8s is still the best at finding unseen changes in animal detection.

Overall Performance

YOLOv8s is the best-performing model overall, achieving the highest precision, recall, and F1-scores across all classes in both validation and test datasets. It maintains a strong balance across all metrics, demonstrating excellent generalization from validation to test data, making it a highly reliable choice. Its outstanding performance in fire and animal detection alongside a good performance in human detection confirms its versatility and robustness, the most effective model for diverse object detection scenarios.

Faster R-CNN with MobileNetV3 is the weakest model overall, especially when it comes to human and animal detection. The model also performs poorly in some fire detection tasks, and its inconsistency and lack of generalization across datasets greatly reduce its usability in real-world situations. The model's poor performance on smaller objects (such as humans and animals) contrasts with its relatively better detection of fire, which appears as a larger object in the dataset. This suggests that Faster R-CNN is particularly weak at detecting small-scale instances, making it unreliable for tasks requiring high precision on smaller objects.

SSD with VGG16 moderate performance overall, it is very good at fire detection, while human and animal detection are weaker. The model's restrictions become even more evident at tougher IoU thresholds (mAP@0.75 and mAP@0.5:0.95), which affects its ability to make high-precision detections. It maintains a certain level of stability between training and validation sets but cannot match the overall effectiveness of YOLOv8s and EfficientDet D1, resulting in decreased suitability for complicated object detection problems.

EfficientDet D1 is a powerful second-best model that is particularly good in a human and fire detection task, and in animal detection it performs at a moderate level. It generalizes well across datasets, although its performance at stricter IoU thresholds is slightly less stable than YOLOv8s. Nonetheless, EfficientDet D1 is particularly effective in scenarios where human and fire detection are a priority, making it a viable alternative when precision in these categories is crucial.

Optimal Model Selection

First, *YOLOv8s is a run-up model that performs best in all metrics and hence is an ideal solution for a thermal dataset*. It delivers stable and balanced detection for all classes. *EfficientDet D1* is a strong alternative for those cases where human and fire detection have priority, whereas *SSD with VGG16* and *Faster R-CNN with MobileNetV3* are less consistent but still functional. This comparison identifies *YOLOv8s* as the most versatile and trustworthy model for fire, human, and animal detection tasks.

4.4 Optimal Confidence Threshold Selection for Each Dataset Using the Best Model

4.4.1 Introduction

The choice of a suitable confidence threshold is a very significant decision in the object detection problems. It directly affects the precision-recall ratio and, as a result, the quality of the model deployed in the system. A very low threshold may cause the detection of false positives too much, whereas a high threshold can lead to the hiding of correct positive detections. To solve this problem, in this part, a search for the optimal confidence thresholds is carried out for both the RGB and Thermal datasets, using the best-performing model identified in previous evaluations (YOLOv8s).

Through the use of F1-score measure, as a function of a confidence threshold, the point is found at which the F1-score is at its peak, symbolizing the most efficient balance between the precision and recall. This approach guarantees that the model functions under circumstances which provide the highest detection capability for each dataset. The chosen confidence levels are then enforced during the real-time inference, thus guaranteeing that the detection remains strong and fair over all the object classes that are relevant, such as fire, smoke, human, and animal. The next sections illustrate the results and justifications of each dataset separately.

4.4.2 Optimal Confidence Threshold Selection for RGB Dataset

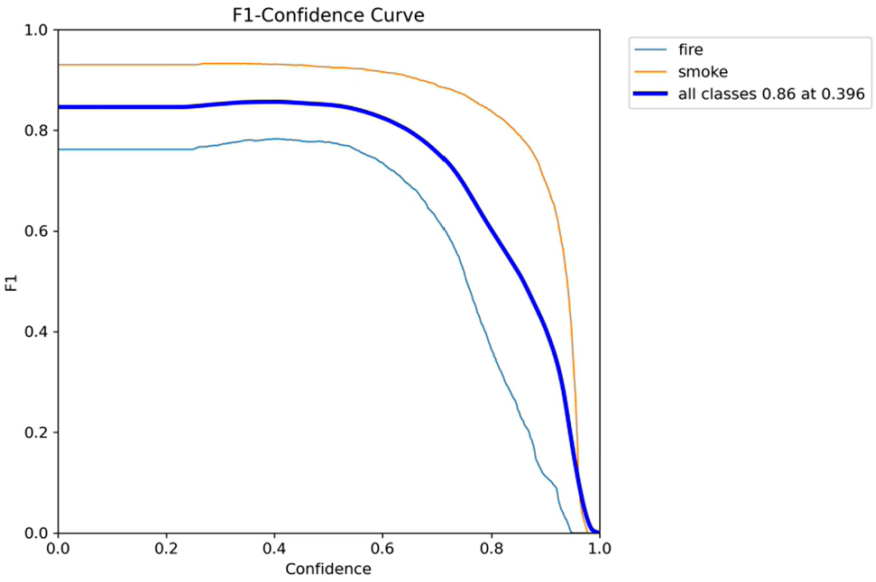


Figure 4.48: F1-Confidence curve on the RGB valid dataset

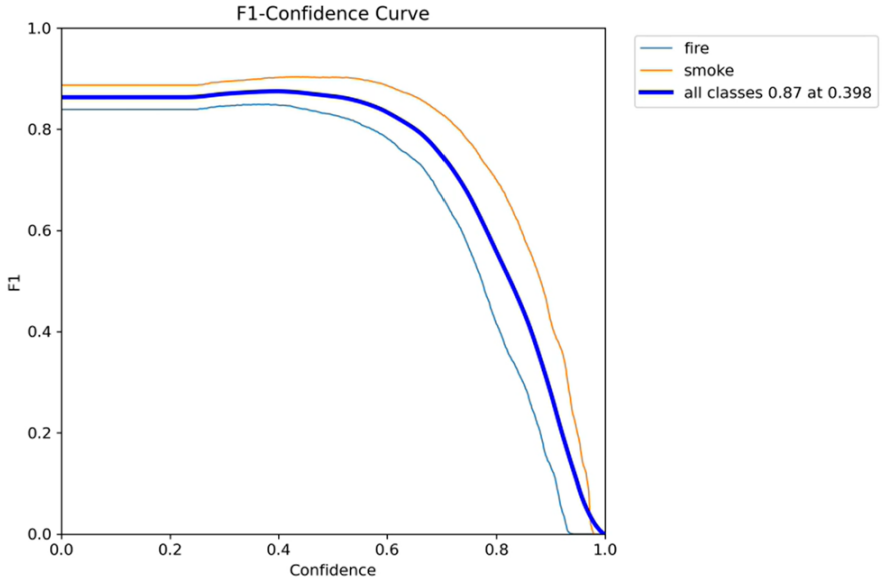
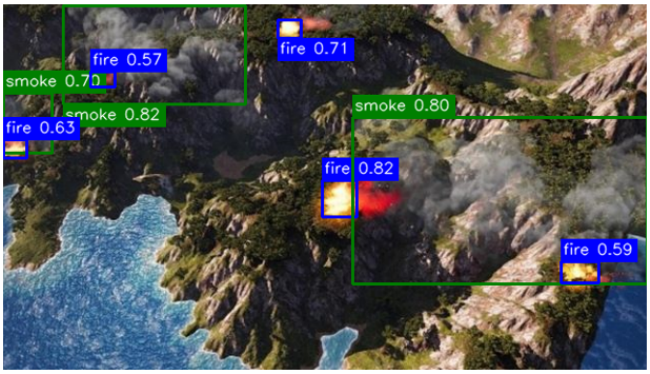
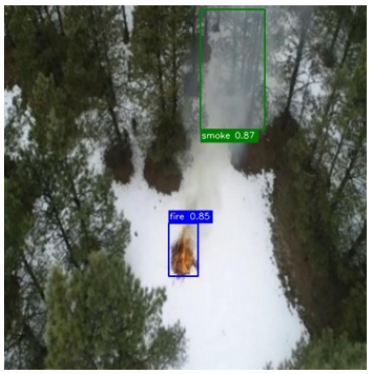


Figure 4.49: F1-Confidence curve on the RGB test dataset

From the two previous plots, the best F1-score for the validation dataset is *0.86* at a confidence threshold of *0.396*, and for the test dataset, it is *0.87* at *0.398*. Based on these results, a *confidence threshold of 0.4* and an *NMS threshold of 0.5* will be used for real-time predictions, ensuring a balanced trade-off between precision and recall. In the examples shown below, as well as in the Subsubsection 4.2.5.1, the model will now be tested to evaluate its performance in predicting fire and smoke classes.

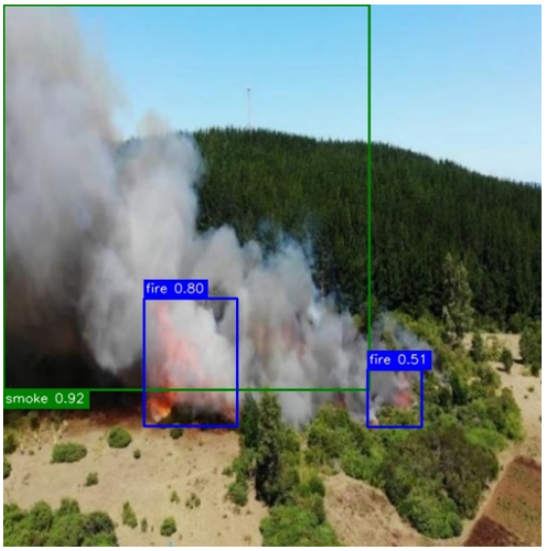




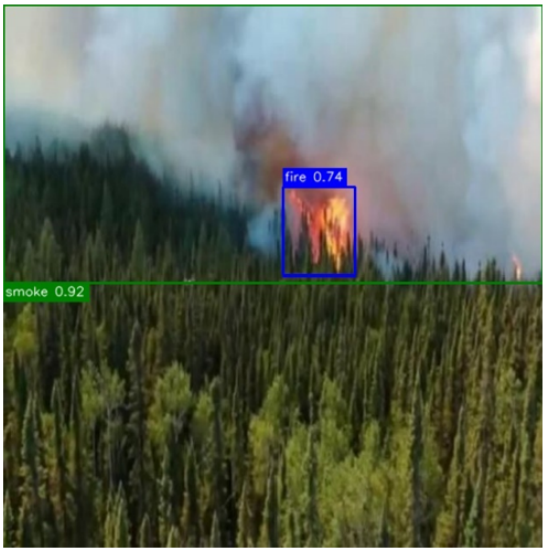

smoke 0.93
fire 0.89


smoke 0.92


smoke 0.56


fire 0.80
fire 0.51
smoke 0.92


fire 0.41
fire 0.73


fire 0.74
smoke 0.92

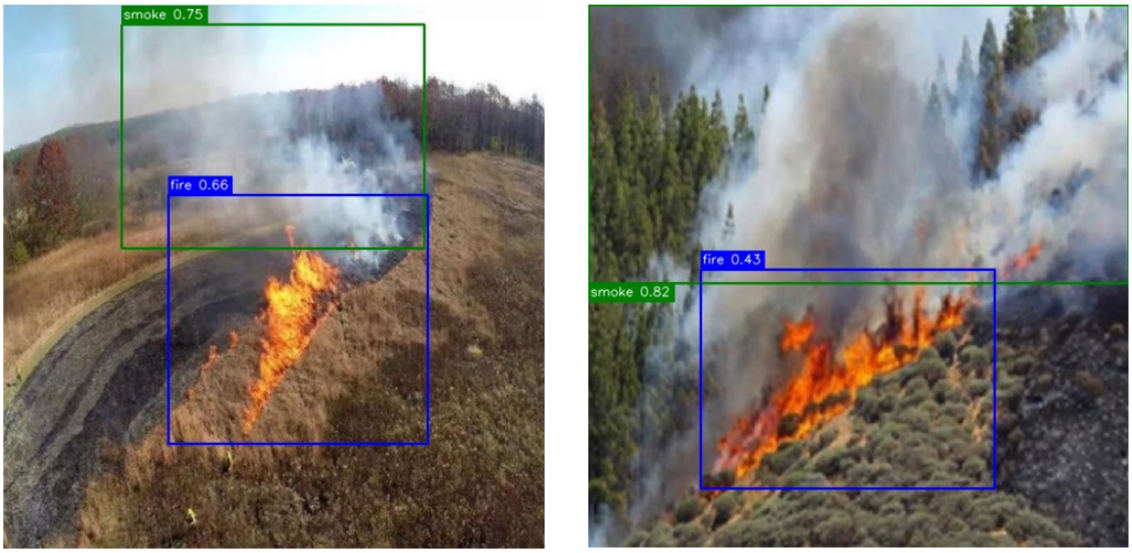


Figure 4.50: Prediction Samples from the RGB dataset

4.4.3 Optimal Confidence Threshold Selection for Thermal Dataset

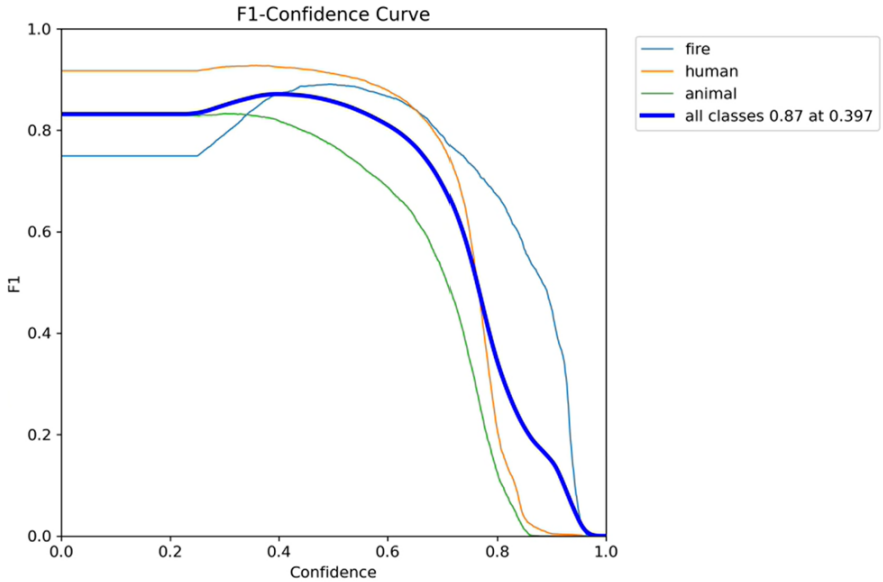


Figure 4.51: F1-Confidence curve on the Thermal valid dataset

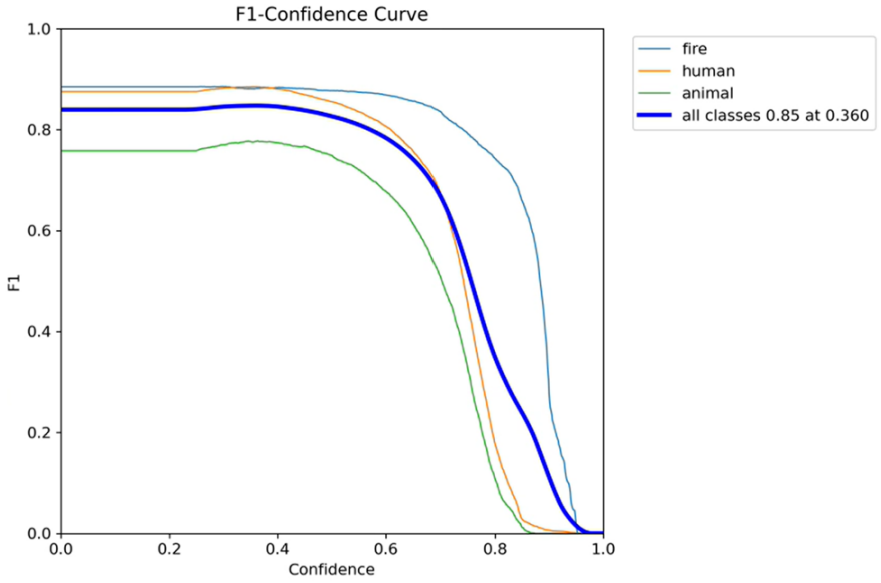
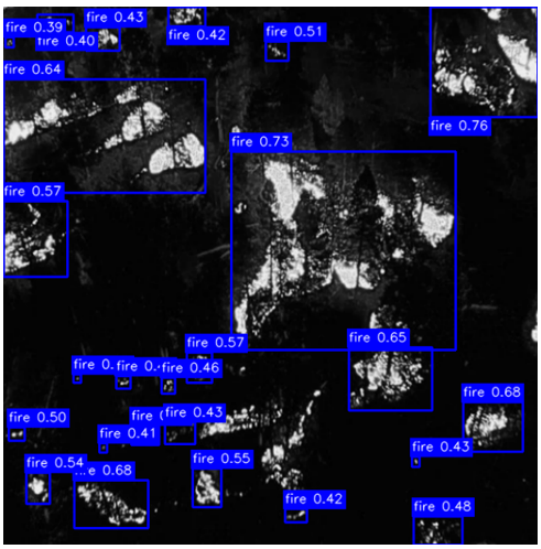
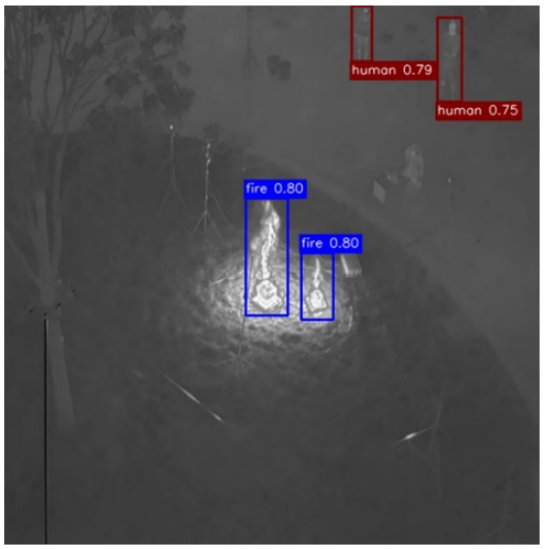
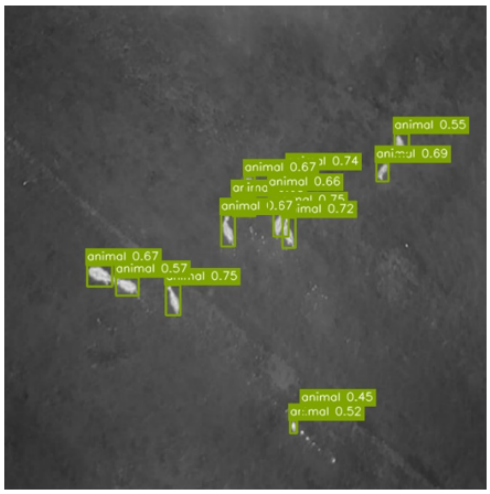


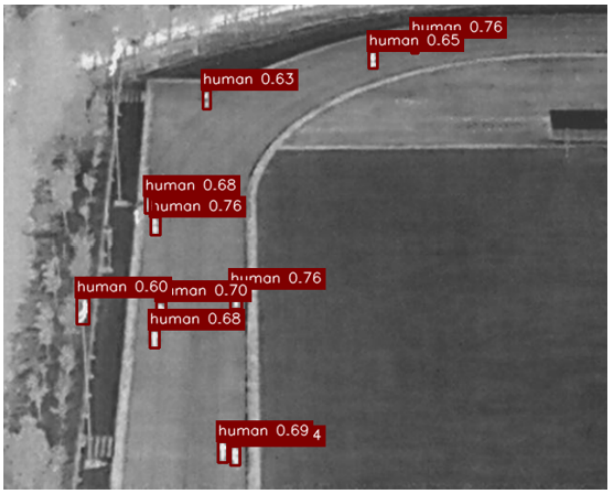
Figure 4.52: F1-Confidence curve on the Thermal test dataset

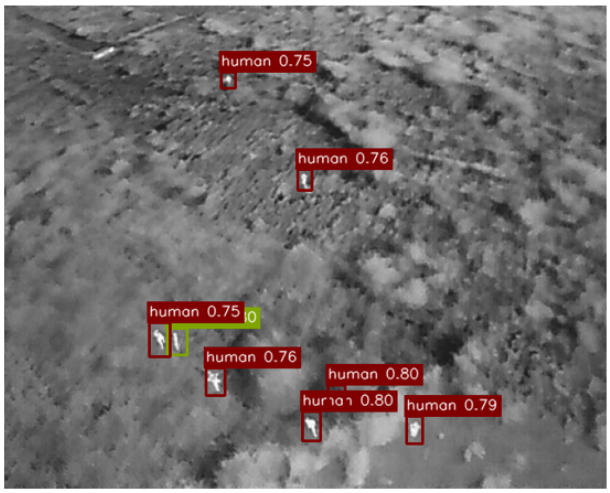
From the two previous plots, the best F1-score for the valid dataset is *0.87* at a confidence threshold of *0.397*, and for the test dataset, it is *0.85* at *0.36*. Based on these results, a *confidence threshold of 0.38* and an *NMS threshold of 0.5* will be used for real-time predictions, ensuring a balanced trade-off between precision and recall. In the examples shown below, as well as in the Subsubsection 4.2.5.2, the model will now be tested to evaluate its performance in predicting fire, human and animal classes.



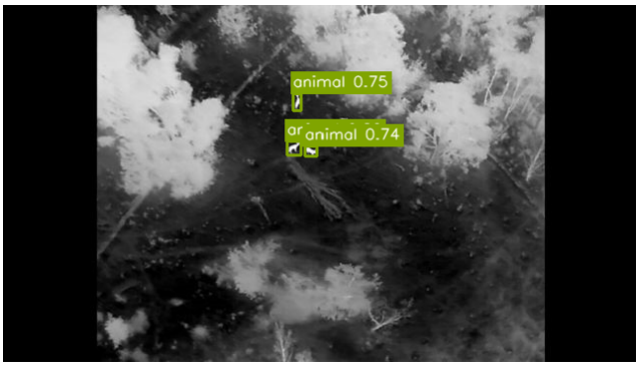



animal 0.55
animal 0.67
animal 0.66
animal 0.72
animal 0.67
animal 0.57
animal 0.75
animal 0.45
animal 0.52

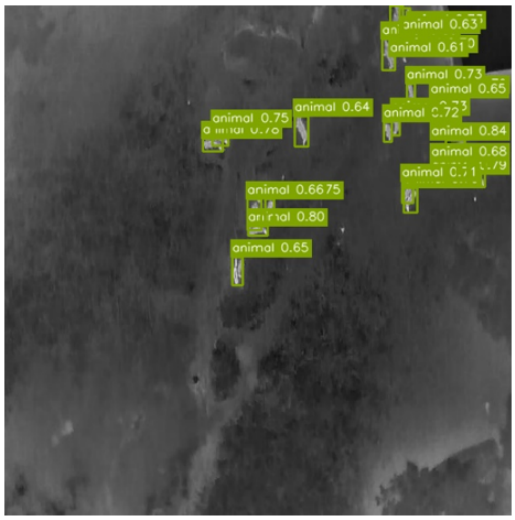

human 0.76
human 0.65
human 0.63
human 0.68
human 0.76
human 0.60
human 0.70
human 0.68
human 0.69

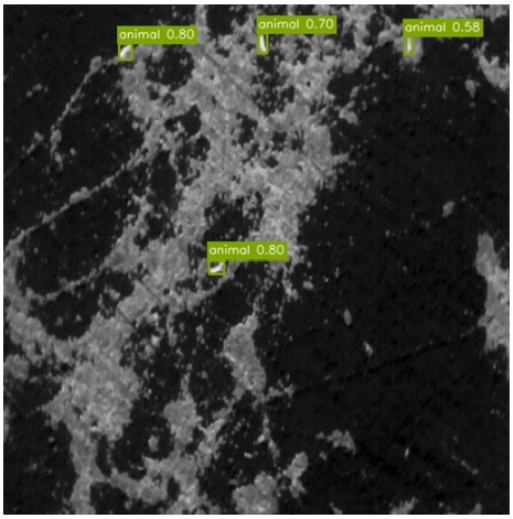

human 0.75
human 0.76
human 0.75
human 0.76
human 0.80
human 0.80
human 0.79

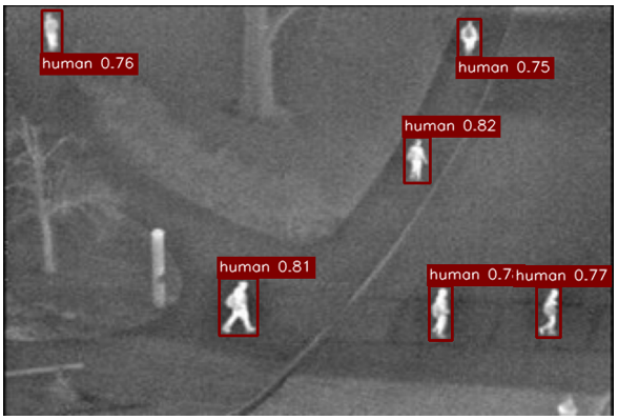

fire 0.85

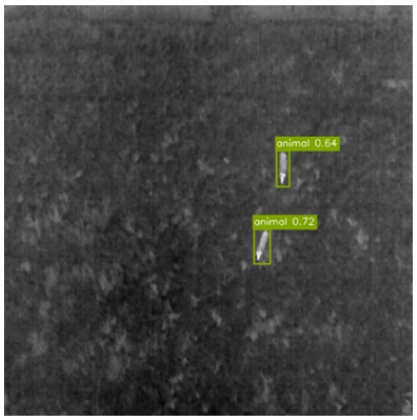

animal 0.75
animal 0.74

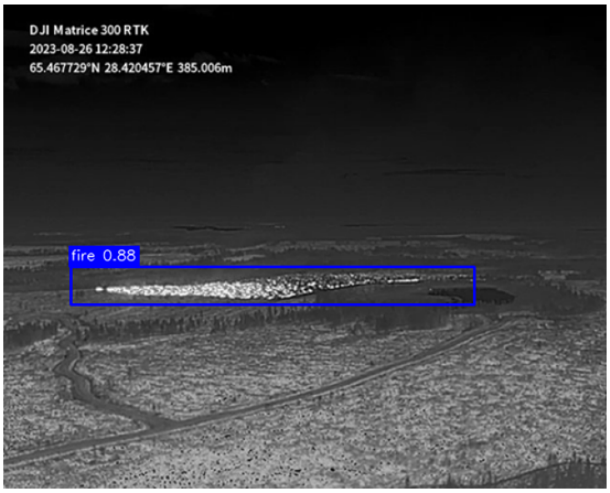

human 0.74
human 0.76
human 0.88
animal 0.72
animal 0.76

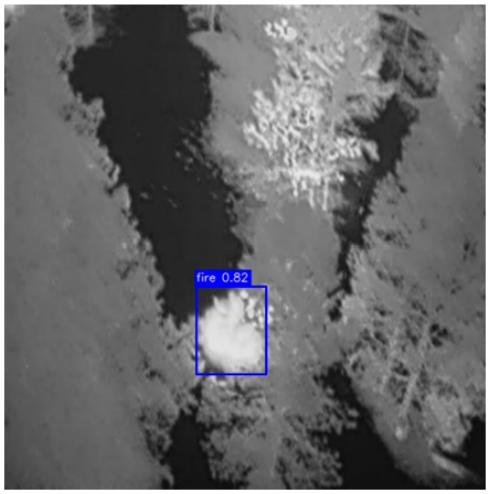

animal 0.64
animal 0.75
animal 0.84
animal 0.68
animal 0.6675
animal 0.80
animal 0.65


animal 0.80
animal 0.70
animal 0.58
animal 0.80


human 0.76
human 0.75
human 0.82
human 0.81
human 0.77


animal 0.64
animal 0.72


DJI Matrice 300 RTK
2023-08-26 12:28:37
65.467729°N 28.420457°E 385.006m
fire 0.88


fire 0.82

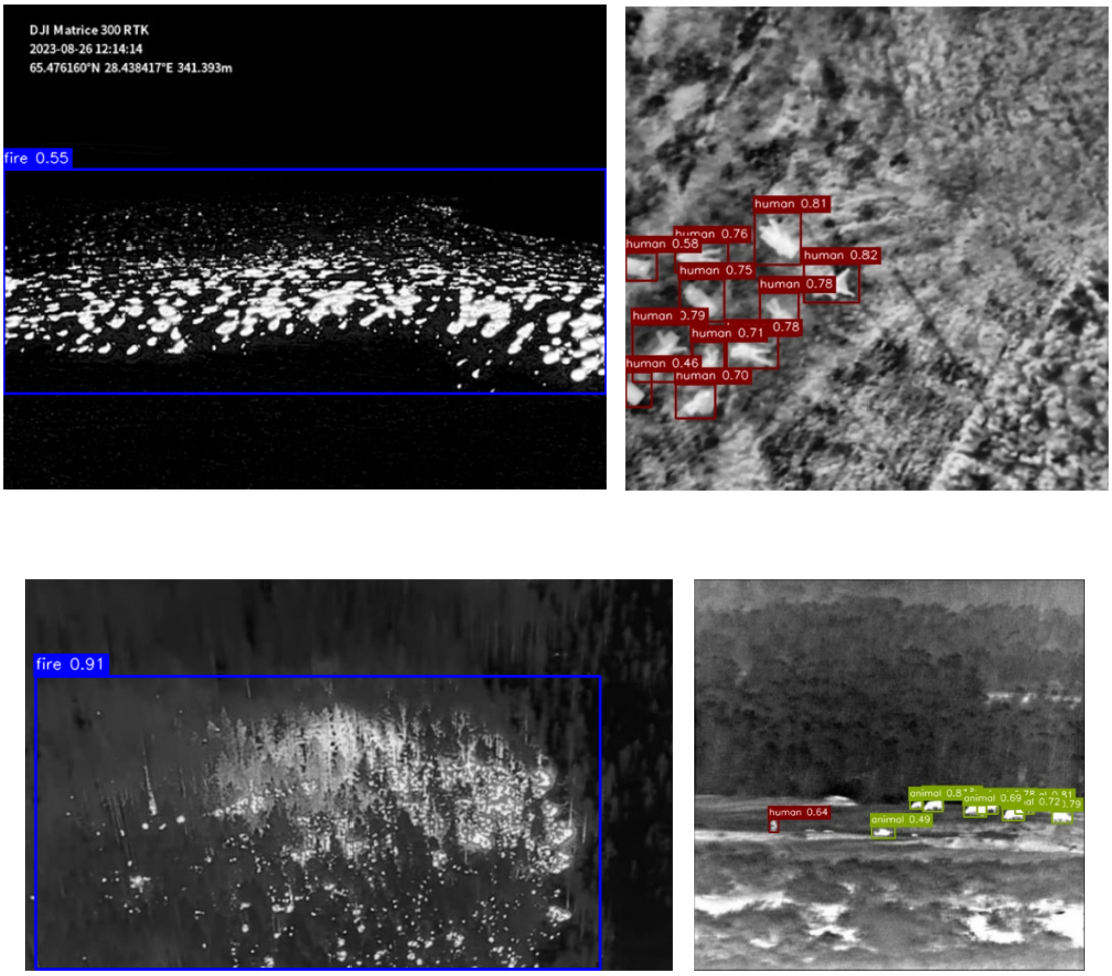


Figure 4.53: Prediction Samples from the Thermal dataset

4.5 TensorRT-Based Optimization of the Best-Selected Model

4.5.1 Introduction

High-speed inference has to be combined with accurate object detection if the problem that detection of objects transformations from research departments to the use in real life is addressed. The deployment pipeline must be optimized for both performance and efficiency in order to satisfy these requirements. TensorRT, a software library for accelerating inference launched by NVIDIA, is the one that enables this by adjusting deep learning models, such as YOLOv8s, for fast inference on NVIDIA GPUs. y leveraging TensorRT's powerful suite of optimizations, including precision reduction, layer fusion, and memory reuse, it is possible to significantly accelerate inference while minimizing computational and energy overhead. These capabilities are of particular importance for real-time applications such as UAV-based

wildfire surveillance, where rapid on-the-fly inference with limited computational resources is required.

This section initiates a deep evaluation of the performance of TensorRT-tuned YOLOv8s models on two distinct hardware platforms: the high-end *NVIDIA GeForce RTX 4090* and the embedded *Jetson Orin Nano*. The RTX 4090 presents the topmost performance that can be possibly achieved in the best conditions, whereas the Jetson Orin Nano allows for a more practical edge deployment situation with stringent power and memory provisions. Both FP16 and INT8 precision formats are considered in order to evaluate the accuracy-efficiency trade-off. This analysis points out the best setting for using the YOLOv8s models to perform multi-class detection tasks (fire, smoke, human, animal) with the aid of RGB as well as thermal camera inputs in real-time environments.

4.5.2 Background

4.5.2.1 Overview of TensorRT

TensorRT [68] is a high-performance inference engine developed by NVIDIA to accelerate the deployment of deep learning models on NVIDIA GPUs. Built on the CUDA framework, it is specifically designed to maximize inference speed while minimizing resource usage. TensorRT can achieve significant performance gains, often speeding up inference by 4 to 5 times for real-time and embedded applications. In some cases, it has been reported to provide up to 40 times faster inference compared to running models solely on a CPU.

4.5.2.2 Optimization Techniques Employed by TensorRT

TensorRT uses a series of sophisticated techniques to enhance the speed and efficiency of deep learning model inference.

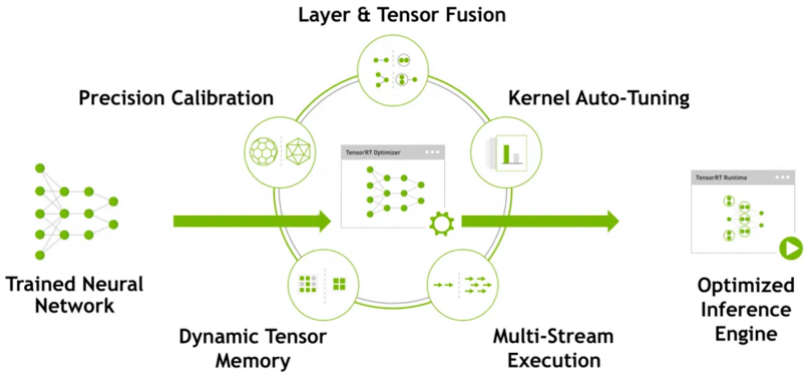


Figure 4.54: TensorRT functionality [68]

Below are the five primary optimizations:

1. Precision Reduction for Weights and Activations

Deep learning models usually learn with parameters and activations in FP32 (32-bit floating point) precision format. TensorRT is a tool that reduces the precision to FP16 or INT8, thus it saves computation time and memory. Moving to FP16 typically minimally changes accuracy because models are structured to keep important features for inference and forget unnecessary noise.

FP32 (Single Precision)

FP32 representation consists of:

- *1 bit* to indicate the sign (positive or negative).
- *8 bits* for the exponent, using base 2 to represent the range of values.
- *23 bits* for the significand (also called the fraction or mantissa), which represents the value after the decimal point.

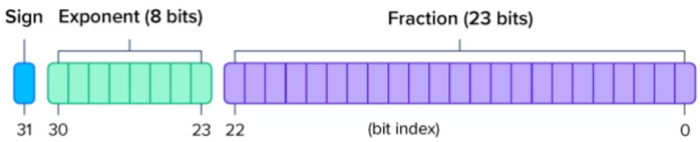


Figure 4.55: FP32 representation [69]

FP16 (Half Precision)

FP16 representation consists of:

- *1 bit* to denote the sign (positive or negative).
- *5 bits* for the exponent, using base 2 to define the range of values.
- *10 bits* for the significand (also known as the fraction or mantissa), which represents the portion of the value after the decimal point.

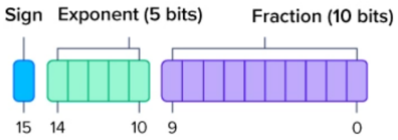


Figure 4.56: FP16 representation [69]

In the case of the conversion to *INT8*, there is a need for extra attention as the limited range of INT8 (from -127 to $+127$) can cause the weights to overflow and the model to lose accuracy. TensorRT solves this problem by introducing the scaling factors that represent the FP32 values in the INT8 range. The method chooses a suitable threshold to keep the model accurate. A small dataset, called the calibration dataset, which is similar to the original data, helps this process go smoothly and reduces the gaps between the FP32 and INT8 distributions.

INT8 Calibration

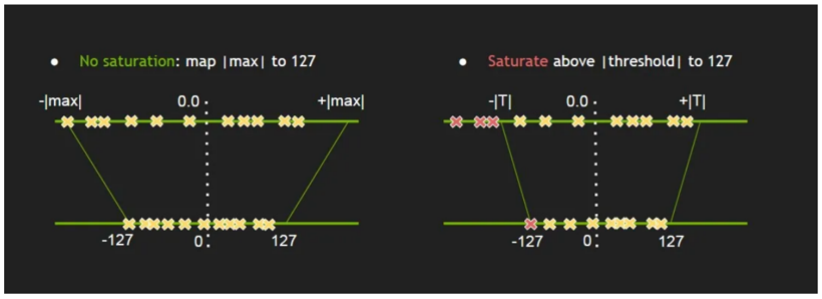


Figure 4.57: INT8 calibration [68]

Standard Mapping (Left Figure)

- In the left side of the image, FP32 values are directly mapped linearly to the INT8 range, from -127 to $+127$.
- The mapping is based on the maximum absolute value ($|max|$) of the FP32 distribution. All FP32 values within the range $[-|max|, +|max|]$ are proportionally scaled to fit within the INT8 range.

- *Limitation*: Values outside $|max|$ in FP32 are clipped, which can result in a loss of information. If $|max|$ is poorly chosen, the range may not capture the entire data distribution effectively, leading to accuracy drops.

Threshold-Based Saturation (Right Figure)

- To mitigate the issues with direct mapping, a threshold value ($|T|$) is introduced.
- Instead of mapping all FP32 values up to $|max|$, values are mapped only up to the chosen $|T|$ threshold, and any outliers beyond this range are clamped to the extreme INT8 values (-127 or $+127$).
- This method preserves more of the essential data distribution by ignoring extreme outliers that could distort the mapping process.

Choosing the Optimal Threshold ($|T|$)

- To find the best threshold value, TensorRT uses *KL-divergence (Kullback–Leibler divergence)*. This statistical method measures the difference between the FP32 data distribution and its INT8 representation. The goal is to minimize this divergence to retain the original model's accuracy.
- TensorRT does this through an *iterative search* process (not gradient descent), testing various thresholds to find the one that minimizes the difference.

Calibration Data

- A small representative dataset, referred to as the *calibration dataset*, is used for this process. The calibration dataset will be the valid dataset of the original dataset.
- The dataset is used to calculate activation histograms, which provide a distribution of FP32 values. These histograms help determine the optimal threshold for mapping FP32 to INT8 while minimizing accuracy loss.
- The calibration algorithm that will be used below is `ENTROPY_CALIBRATION_2`. This uses a saturated threshold for the mapping from FP32 to INT8.

2. Layer and Tensor Fusion

TensorRT minimizes computational overhead through the merger of layers or operations that can be carried out in a single processing step. As an example, a layer that has a similar characteristics such as input size and filter configuration can be merged into one kernel. It goes a step further by not only saving memory but also increasing execution speed as it eliminates the repeated loading and saving of intermediate data.

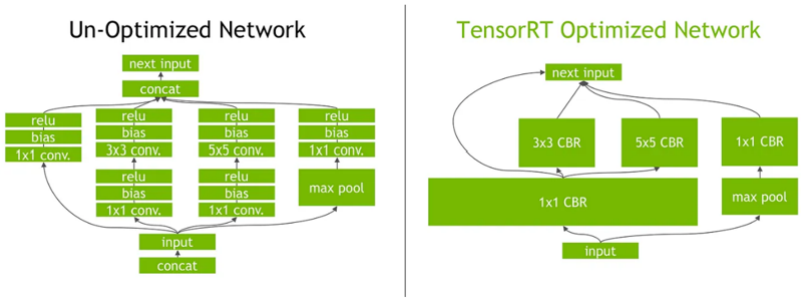


Figure 4.58: Layer and tensor fusion [68]

3. Kernel Optimization and Auto-Tuning

TensorRT improves the efficiency of layers execution by picking the best implementation that is most suitable for the target GPU based on available hardware resources. One case is the convolution operations that can be implemented in various ways, and TensorRT is the one that figures out the quickest route to follow. Also, it adjusts parameters such as batch size to get even better results.

4. Efficient Memory Management

TensorRT deals with memory as efficiently as possible by providing memory only for the tensors that are actually being used at a particular time. Whenever a tensor's job has been completed, its memory is freed up and can be used for other calculations. This scheme of flexible memory greatly cuts down the overall memory usage and helps in faster execution.

5. Parallel Execution of Streams

TensorRT uses CUDA streams technology of NVIDIA, which is the capacity of the GPU to run multiple kernels simultaneously, to perform multiple tasks at the same time. Therefore, processing multiple input streams in parallel, it achieves higher throughput and fully utilizes GPU resources, which is perfect for scenarios that need real-time responses.

Thus, TensorRT has the potential of deploying deep learning models at a higher speed and with more efficiency by utilizing these innovative techniques, hence, it becomes a must-have

tool in the real-time applications and for devices which have limited computational resources.

4.5.2.3 Overview of NVIDIA Jetson and NVIDIA JetPack

NVIDIA Jetson is a series of embedded computing platforms that are explicitly built to allow AI-powered applications at the edge. The Jetson modules based on NVIDIA's GPU architecture provide the compact and energy-efficient form factor with powerful parallel processing capabilities. These devices support ARM64 architecture and are designed to efficiently run deep learning and computer vision tasks locally, without the need for continuous cloud connectivity. Jetson boards are widely used in robotics, autonomous systems, and industrial automation as they are capable of delivering low-latency, real-time AI inference, thus making them suitable for performance-critical and power-constrained environments.

The NVIDIA JetPack SDK is the official development platform for Jetson modules, thus providing a comprehensive set of tools and libraries to accelerate AI application development. This effort supplies a complete setup including Jetson Linux with bootloader, Linux kernel, Ubuntu-based desktop interface, and a large array of libraries optimized for GPU computing, multimedia processing, computer vision, and graphics. In addition to this, JetPack also gives developer tools, samples, and documentation both for the host as well as Jetson device, thus enabling efficient prototyping and deployment. Furthermore, it is compatible with high-level SDKs such as DeepStream for video analytics, Isaac for robotics, and Riva for conversational AI. *JetPack 6.1* is used in this research work for developing and deploying the optimized inference pipeline on the Jetson platform.

4.5.3 Real-Time Inference Performance Evaluation for YOLOv8s Optimized with TensorRT FP16-INT8

The *storage size* of the *FP16 model is 24.4MB*, whereas the *FP32 model is 21.5MB*. This discrepancy occurs because the *TensorRT model is converted from an ONNX model, which is 42.5MB*. Despite this, the FP16 storage size is indeed reduced compared to the original ONNX model. Furthermore, the *INT8 model has a storage size of 15MB*, showing an additional reduction from the FP16 model.

Table 4.24: Comparison of inference performance across models on NVIDIA GeForce RTX 4090

Model	GPU Memory Usage (MB)	GPU Temp Increase (°C)	GPU Power (W)	CPU Usage (%)	RAM Usage (MB)	FPS	Latency (ms)
YOLOv8s FP32 (original)	174.06	4	59.83	0.6	572.06	482	2.08
YOLOv8s FP16	40	1	12.92	0.2	41.38	605	1.65
YOLOv8s INT8	**20**	**0**	**7.88**	**0**	**9.34**	**736**	**1.36**

Similarly to Subsection 4.3.5, the FP32 model is used as a reference. From Table 4.24 we see that the other two models (FP16 and INT8) are still more efficient in terms of resource footprint and, in particular, GPU and CPU usage. The most significant improvement is in the reduction of GPU power consumption, which is in fact one of the most important factors for edge devices such as drones that must consume the minimum amount of power to be able to run longer. Besides that, the increased FPS is like a bonus because in situations where the model is to be deployed at an edge device, the FPS of the model will inevitably drop due to hardware limitations. However, starting with a higher FPS ensures that even after deployment, the model will still achieve a better frame rate compared to the original FP32 model, leading to improved real-time performance.

Table 4.25: Comparison of inference performance across models on NVIDIA Jetson Orin Nano

Model	Concurrent Models	GPU Memory Usage (MB)	RAM Usage (MB)	FPS	Latency (ms)
YOLOv8s FP32 (original)	1	239	261	23.19	43.13
YOLOv8s FP32 (original)	2	413	270	12.95	77.2
YOLOv8s FP16	1	138	217	56.09	17.83
YOLOv8s FP16	2	198	229	32.11	31.14
YOLOv8s INT8	1	**137**	**213**	**79.05**	**12.65**
YOLOv8s INT8	2	**196**	**218**	**42.88**	**23.32**

As shown in Table 4.25, the performance comparison between FP32, FP16, and INT8 models on the Jetson Orin Nano highlights the significant efficiency gains achieved through TensorRT optimization. The FP32 model, used as a reference baseline, demonstrates a higher usage of GPU memory and RAM, along with increased latency and lower FPS. In contrast, the FP16 and INT8 models show substantial improvements in all measured aspects. Specifically, the INT8 model achieves the highest frame rate (79.05 FPS with one model and 42.88 FPS with two models), while maintaining the lowest latency (12.65 ms and 23.32 ms, respectively), making it ideal for applications demanding high throughput.

Furthermore, both FP16 and INT8 models consume significantly less GPU memory and RAM, enabling the deployment of multiple concurrent models without exceeding hardware constraints. In particular, the evaluation with two concurrent models simulates the actual deployment scenario of this system, where two models, one for the RGB camera and one for the thermal camera, are required to run simultaneously. Even under this dual-model setup, the FP16 and INT8 models maintain acceptable latency and FPS, validating their suitability for real-time inference on edge devices such as the Jetson Orin Nano.

Additionally, GPU temperature increase and power consumption remain very low across all models, which is crucial for maintaining device longevity and operational efficiency on embedded systems. CPU usage is also minimal, reaching at most 10% in the most demanding scenarios, further strengthening the feasibility of deploying these models in power-constrained environments such as UAVs.

4.5.4 Performance Analysis Based on Detection Metrics

4.5.4.1 Performance Analysis of Optimized YOLOv8s in the RGB Dataset

The evaluation time on the test dataset was *14.1 seconds for the FP32 model*, *9.03 seconds for the FP16 model*, and *7.94 seconds for the INT8 model*.

Table 4.26: Val/Test metrics on the RGB dataset with FP16 model

Class	split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
overall	Val	0.897	0.825	0.860	0.898	0.689	0.664
overall	Test	0.902	0.852	0.876	0.912	0.655	0.620
fire	Val	0.848	0.748	0.795	0.838	0.494	0.505
fire	Test	0.885	0.811	0.846	0.881	0.589	0.555
smoke	Val	0.947	0.902	0.924	0.958	0.883	0.824
smoke	Test	0.918	0.894	0.906	0.943	0.721	0.685

Table 4.27: Val/Test metrics on the RGB dataset with INT8 model

Class	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
overall	Val	0.882	0.821	0.851	0.882	0.668	0.646
overall	Test	0.888	0.828	0.857	0.889	0.624	0.593
fire	Val	0.819	0.725	0.769	0.812	0.467	0.480
fire	Test	0.868	0.770	0.816	0.848	0.546	0.513
smoke	Val	0.946	0.918	0.931	0.952	0.870	0.813
smoke	Test	0.908	0.886	0.897	0.930	0.702	0.674

As observed, the FP16 model shows only a 0–1% decrease in overall performance, while the INT8 model experiences a 1–4% decrease. The optimal confidence threshold for achieving the best F1 score remains 0.4 for the FP16 model, whereas for the INT8 model, it is 0.32. This indicates that the INT8 model is generally less confident in detecting fire and smoke correctly.

4.5.4.2 Performance Analysis of Optimized YOLOv8s in the Thermal Dataset

The *evaluation time* on the test dataset was *31.65 seconds for the FP32 model*, *10.99 seconds for the FP16 model*, and *9.69 seconds for the INT8 model*.

Table 4.28: Val/Test metrics on the Thermal dataset with FP16 model

Class	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
overall	Val	0.917	0.809	0.859	0.895	0.689	0.637
overall	Test	0.880	0.826	0.853	0.883	0.581	0.549
fire	Val	0.887	0.787	0.834	0.864	0.807	0.760
fire	Test	0.945	0.857	0.899	0.932	0.726	0.636
human	Val	0.948	0.899	0.923	0.951	0.707	0.631
human	Test	0.900	0.862	0.880	0.901	0.509	0.528
animal	Val	0.915	0.741	0.819	0.870	0.553	0.521
animal	Test	0.796	0.761	0.778	0.815	0.506	0.484

Table 4.29: Val/Test metrics on the Thermal dataset with INT8 model

Class	Split	Precision	Recall	F1-score	mAP@0.5	mAP@0.75	mAP@0.5:0.95
overall	Val	0.927	0.783	0.849	0.875	0.664	0.614
overall	Test	0.898	0.739	0.811	0.836	0.574	0.527
fire	Val	0.918	0.787	0.848	0.882	0.842	0.775
fire	Test	0.929	0.788	0.853	0.879	0.771	0.632
human	Val	0.946	0.897	0.921	0.939	0.660	0.597
human	Test	0.923	0.828	0.873	0.889	0.493	0.509
animal	Val	0.916	0.665	0.771	0.803	0.491	0.469
animal	Test	0.841	0.603	0.702	0.740	0.457	0.438

As we can see, the FP16 model exhibits incremental detection improvements in certain metrics, which are attributed to TensorRT optimizations, GPU acceleration, and small floating-point precision differences. These optimizations improve the inference efficiency, which in turn results in small performance gains. On the other hand, in some instances, the FP16 model shows a minor decline of about 0–2%, whereas the INT8 model reveals a more substantial reduction of 2–6%. The best F1 score can be obtained by setting the confidence threshold to 0.38 for the FP16 model, while for the INT8 model, it is 0.25. This indicates that the INT8 model is generally less confident in correctly detecting fire, humans, and animals.

4.5.5 Conclusions

Based on the evaluation conducted on both *NVIDIA RTX 4090* and *Jetson Orin Nano*, the *FP16 model* is the preferred choice due to its balance between *accuracy and efficiency*. It has *only a slight performance drop (0–2%)* compared to FP32 while improving *GPU and CPU resource utilization*, *FPS*, and *power consumption* substantially. This makes it ideal for real-time applications, particularly on edge devices such as drones where power and thermal constraints are critical, as shown in the Jetson Orin Nano benchmarks.

On the other hand, the *INT8 model*, despite the fact that it shows a larger decrease in resource consumption, it also suffers from a more evident performance decay (*2–6% drop in detection accuracy*) and has a lower confidence threshold for correct detections. It thus becomes *less reliable for critical tasks*, such as detecting *fire, humans, and animals with the thermal camera*, and *fire and smoke with the RGB camera*, especially in those situations where high precision and recall are necessary.

FP16 will be used as the primary model to ensure high detection accuracy while still benefiting from performance optimizations. *If further GPU optimizations are required*, the *INT8 model can be used as an alternative*, prioritizing *resource efficiency over detection accuracy* in scenarios where power consumption is a critical constraint, such as on embedded edge platforms like the Jetson Orin Nano.

Chapter 5

System Implementation

5.1 Introduction

This chapter outlines the theoretical concepts and practical steps that are the foundation of the suggested wildfire detection system utilizing RGB and thermal cameras on a simulated UAV. Initially, the chapter is equipped with fundamental background necessary for comprehension of the system which includes the core concepts of the ROS 2 framework, camera calibration principles, and object tracking algorithms that are related to UAV-based applications.

The chapter then moves on to describe the procedure that was used to geometrically align the RGB and thermal camera feeds which is the calibration and registration procedure. Accurate alignment between the two modalities is crucial for reliable detection and fusion, this section of the chapter, therefore, imparts the use of a checkerboard-based calibration method, homography computation, and cropping for field-of-view consistency as the steps that have been carried out to achieve that.

The last section of the chapter is an account of the full ROS 2 node architecture which is the implementation of the wildfire detection pipeline. It reveals the high-level system architecture of the custom ROS 2 messages, the software packages and the detailed description of each node's functionality. The section also tells us how to start the system and actually run it by showing all of the components working together via ROS 2 synchronization and integration.

Overall, this chapter serves as the core of the system design, bridging foundational knowledge with the concrete software and simulation architecture implemented in this work.

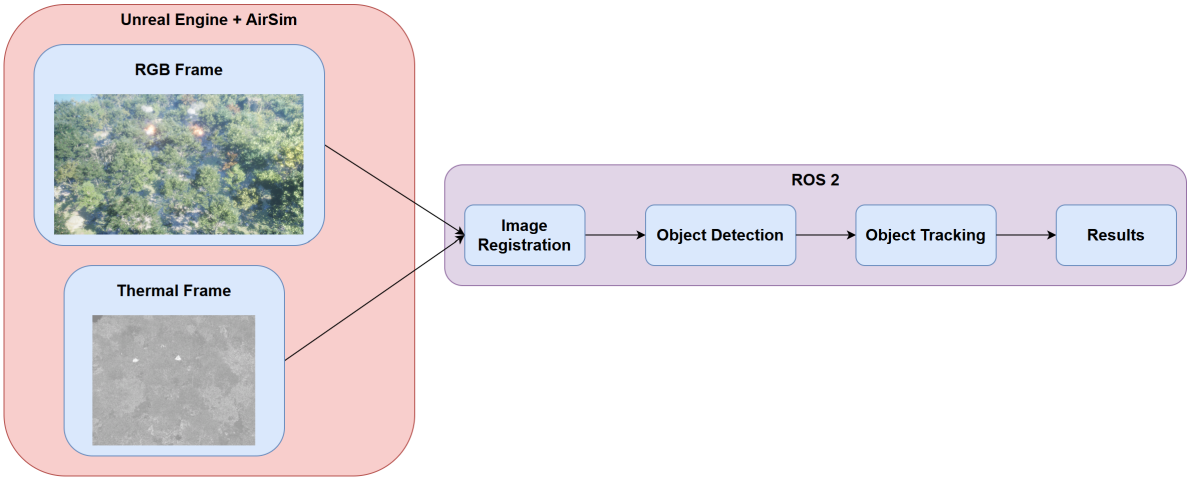


Figure 5.1: System architecture

5.2 Background

5.2.1 Fundamental Concepts in ROS 2 Communication

ROS 2 (Robot Operating System 2) is a middleware framework that facilitates distributed communication among modular components in a robotic system. Understanding the core communication primitives, namely topics, publishers, subscribers, messages, services and actions, is essential to comprehend how the UAV-based wildfire detection system operates.

Topics, Publishers and Subscribers

Topics are one-way communication channels designed for continuous data streaming between nodes. A node that sends data is referred to as a *publisher*, while a node that receives data is a *subscriber*. The publisher does not need to know which subscribers exist, and vice versa. This decoupled design allows for scalable and flexible architectures. In this system, to give an example, the sensor data collector node publishes RGB and thermal image streams along with GPS data to dedicated topics, which are then subscribed to by the preprocessing node.

Messages

Messages are the data structures that travel over topics. ROS 2 offers standard message types (ex., `sensor_msgs/Image`, `geometry_msgs/Point`), but custom messages can be created by systems in order to correspond with the application-specific needs. In this work, custom message types like `ImageWithGPS` and `DetectionWithGPS` are introduced to represent enriched sensor data and detection results, including image frames,

labels, confidence scores, bounding boxes, and GPS coordinates.

Services

Services enable synchronous, request-response communication between two nodes. A service is defined by a request and a corresponding response type. Unlike topics, which are used for continuous data streams, services are a perfect choice for one-time queries or control commands. Although the existing system implementation is not services-oriented very much, services are suitable for tasks such as triggering a manual override, requesting UAV status, or initiating a reset.

Actions

Actions extend the ROS 2 service model to enable long-running, interruptible jobs. They give clients the possibility to send a goal, get repetitive feedback while the execution is in progress, and receive the final result or the confirmation of cancellation. In UAV applications, actions are typically employed for the execution of navigation tasks like for example, setting a waypoint goal and observing the progress of the task in real time.

Even though the existing system implements the waypoint-based path planning for UAV navigation, it does not adopt ROS 2 actions. On the other hand, the system is managing waypoints by direct topic publishing, which is enough for the current implementation of a linear path of a UAV that is predefined. This is, however, a very strong and scalable alternative for future work represented by ROS 2 actions, particularly for more complex navigation scenarios involving dynamic goal updates, feedback monitoring, or return-to-home behavior.

Parameters

ROS 2 nodes additionally can use parameters, which are configurable variables that can be declared and modified at runtime. Parameters are employed to steer the node's behavior without changing the code, for instance, setting confidence thresholds, toggling tracking, or deciding which tracking model to load. These values are most often specified in YAML files and given to launch files at the time of system startup.

Collectively, these basic communication primitives outline the core of the ROS 2 framework and make possible a smooth interaction among different software modules that compose a wildfire detection and UAV control system.

5.2.2 Camera Calibration

Camera calibration is a fundamental procedure in computer vision that establishes the mathematical relationship between the three-dimensional (3D) world and its two-dimensional (2D) image projection. The goal of calibration is to determine the parameters of the camera model that describe how the camera captures visual information from the environment. These parameters are generally divided into two categories: intrinsic and extrinsic.

Intrinsic Parameters

Intrinsic parameters are the internal characteristics of the camera that define how a 3D point in camera coordinates is projected onto the 2D image plane. These parameters include the focal length (in pixels), the coordinates of the principal point (generally near the image center), and the skew coefficient (which is definitely very small in modern cameras). The intrinsic matrix may, in addition, incorporate the effects of pixel scaling in case the pixels of the sensor are not square. The parameters can be represented by the 3×3 camera matrix K, which forms the basis of the pinhole camera model and thus, it is used to project the 3D points into 2D image coordinates.

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

- f_x: focal length in pixels along the x-axis.
- f_y: focal length in pixels along the y-axis.
- s: skew coefficient (models the angle between x and y pixel axes; typically 0 for modern cameras).
- c_x: x-coordinate of the principal point (usually near the center of the image).
- c_y: y-coordinate of the principal point.

Distortion Coefficients

Real-world lenses introduce imperfections that deviate from the ideal pinhole camera model. These imperfections are modeled using distortion coefficients. The most common types of distortion are:

- *Radial distortion*, which causes straight lines to appear curved, especially near the edges of the image (e.g., barrel or pincushion distortion).
- *Tangential distortion*, which occurs when the lens and the image sensor are not perfectly aligned, causing the image to shift asymmetrically.

Distortion coefficients are part of the intrinsic calibration process and are used to correct the raw image, thereby enabling accurate geometric analysis and alignment with other cameras or sensors.

$$\text{Distortion Coefficients} = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}$$

- k_1, k_2, k_3: radial distortion coefficients.
- p_1, p_2: tangential distortion coefficients.

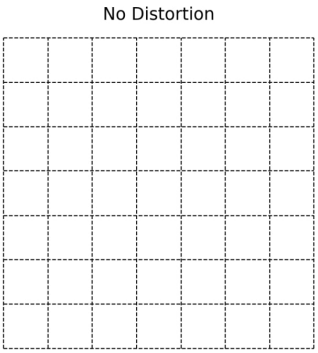


Figure 5.2: No distortion [70]

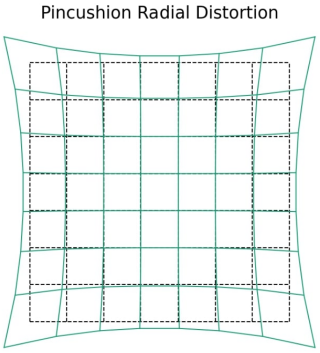


Figure 5.3: Positive (pincushion) distortion [70]

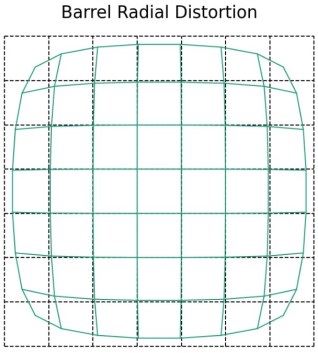


Figure 5.4: Negative (barrel) distortion [70]

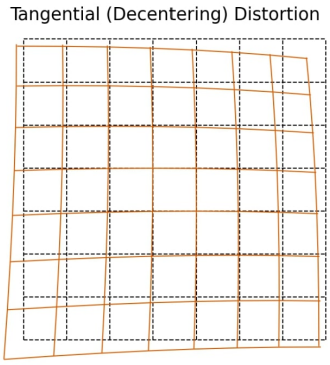


Figure 5.5: Tangential distortion [70]

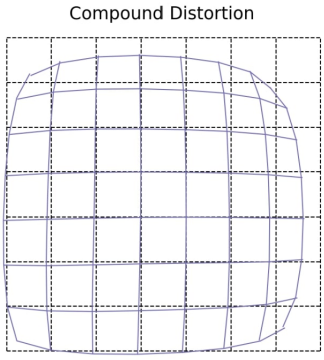


Figure 5.6: Barrel + pincushion + tangential [70]

Let (x, y) be the normalized ideal (undistorted) coordinates, and (x_d, y_d) the distorted coordinates. The distortion is applied as follows:

$$r^2 = x^2 + y^2$$

$$x_{\text{radial}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{\text{radial}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$x_{\text{tangential}} = 2p_1 xy + p_2(r^2 + 2x^2)$$
$$y_{\text{tangential}} = p_1(r^2 + 2y^2) + 2p_2 xy$$

$$x_d = x + x_{\text{radial}} + x_{\text{tangential}}$$
$$y_d = y + y_{\text{radial}} + y_{\text{tangential}}$$

These equations are used to project ideal image points onto the actual distorted image plane. During calibration, the distortion coefficients are estimated so that the inverse of this transformation can be applied to undistort real images.

Extrinsic Parameters

Extrinsic parameters define the camera's position and orientation with respect to a fixed world or object coordinate system. They consist of a 3×3 rotation matrix and a 3×1 translation vector. Together, they describe how to transform 3D points from world coordinates into the camera's coordinate system. These parameters are crucial in multi-camera setups or when aligning camera data with other sensors such as LiDAR.

$$\text{Extrinsic Transformation: } X_c = RX_w + t$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

- R: 3×3 rotation matrix that defines the orientation of the camera with respect to the world coordinate system.

- t: 3×1 translation vector that defines the position of the camera origin in world coordinates.
- X_w: 3D point in world coordinates.
- X_c: 3D point in the camera coordinate system.
- $X_c = RX_w + t$: Transformation equation converting a point from world coordinates to the camera's local coordinate frame.

In summary, camera calibration provides the necessary mathematical tools to translate between the physical environment and image data. By estimating both intrinsic and extrinsic parameters, as well as correcting lens distortions, it enables accurate image rectification, 3D reconstruction, and multi-sensor fusion.

In this thesis, only the distortion coefficients, part of the intrinsic parameters, are used to undistort both RGB and thermal images. Full extrinsic calibration is not performed, as the objective is limited to 2D geometric alignment for multi-modal image registration rather than 3D spatial localization.

5.2.3 Object-Tracking Algorithms

In this thesis, the real-time multi-object tracking system is designed to be modular and supports interchangeable tracking backends. Specifically, the implementation allows the user to select among three state-of-the-art tracking algorithms: *ByteTrack*, *DeepSORT*, and *BoT-SORT-ReID*.

5.2.3.1 ByteTrack

ByteTrack [71] is a real-time multi-object tracking (MOT) algorithm introduced by Yifu Zhang et al. in ECCV 2022 [72]. Unlike traditional tracking approaches that discard low-confidence detection boxes, ByteTrack preserves these boxes and uses them in a secondary association step to enhance tracking robustness.

The core idea of ByteTrack is to retain low-confidence, non-background detections and incorporate them into the data association process. This two-stage association framework as shown in Figure 5.7 is summarized as follows:

- *First Association:* High-confidence detection boxes are matched with existing tracklets (active and recently lost) using Intersection over Union (IoU) or appearance similarity (e.g., cosine distance). The Hungarian algorithm is used to solve the linear assignment problem.

- *Second Association:* Unmatched low-confidence detections are then matched with the remaining unmatched predicted tracklets using a lower threshold. This step allows recovering tracklets lost due to occlusion or temporary appearance degradation.

Tracklets that remain unmatched are marked as *lost*, but are retained for a few frames in case the object reappears. This mechanism enables ByteTrack to handle occlusions and maintain identity consistency over time.

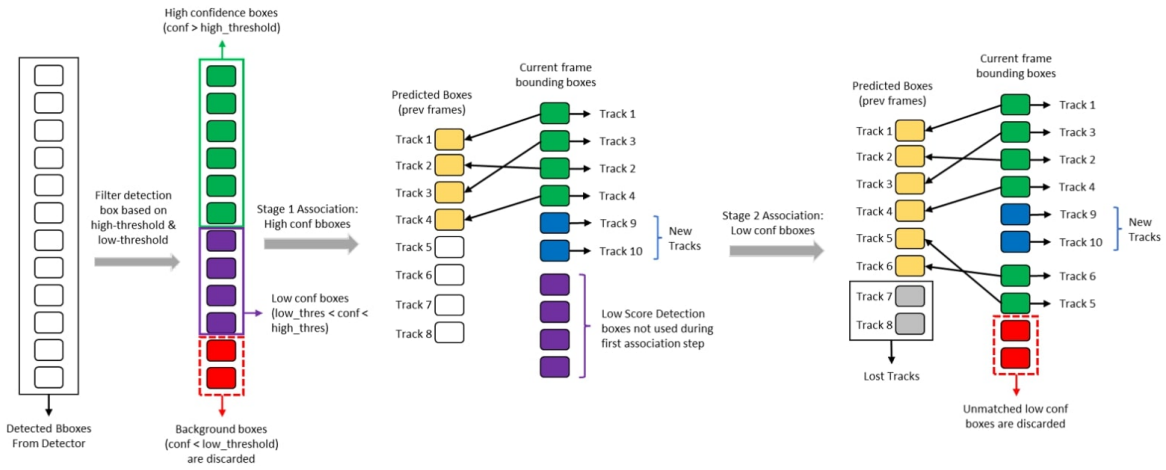


Figure 5.7: ByteTrack algorithm [71]

ByteTrack's universal and simple framework makes it compatible with various object detectors (e.g., YOLO, Faster R-CNN) and appearance matching methods (e.g., DeepSORT, BoT-SORT). Its strong performance and efficiency have led to widespread adoption in modern MOT pipelines.

5.2.3.2 DeepSORT

DeepSORT [73] is a widely used multi-object tracking method that is known for its efficiency in video streams. The algorithm is an improvement upon the SORT (Simple Online and Realtime Tracking) algorithm that uses Kalman filter to do the object tracking.

Deep SORT exploits a deep association metric along with the appearance features that are learned by a convolutional neural network (CNN), therefore it is able to deal with occlusions and temporary disappearance of objects. Thus, the algorithm is very reliable in complicated tracking scenes, e.g. surveillance and self-driving.

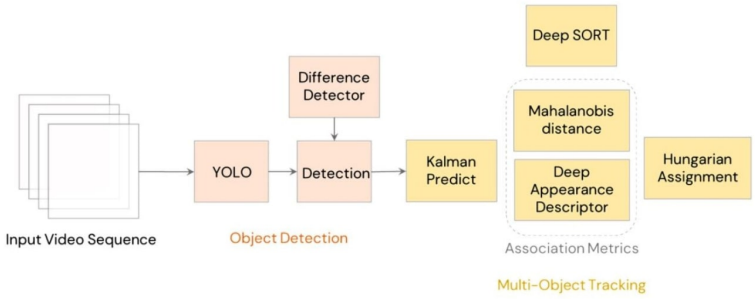


Figure 5.8: DeepSORT architecture [74]

Deep SORT is built on four key components:

- *Detection and Feature Extraction:* The algorithm detects objects firstly, which is usually done by CNNs like YOLO. The algorithm then extracts a high-dimensional appearance descriptor for each detection which represents the visual features of the objects to be matched across frames.

- *Kalman Filter in State Prediction:* In the same manner as SORT, Deep SORT also uses the Kalman filter to figure out the state of an object (position, velocity and acceleration) that is based on its previous state, thus enabling smooth tracking even if the object is hidden for a short time.

- *Data Association with Deep Appearance Metrics:* Contrasting to SORT, Deep SORT joins the appearance feature besides the motion information to get the correct matching. It utilizes a mixture of Mahalanobis distance (for the motion consistency) and cosine similarity (for the appearance matching) in order to associate the detections across frames, by means of the Hungarian algorithm.

- *Track Management:* Deep SORT also introduces some track management mechanisms such as track confirmation (requiring multiple detections to confirm a track) and track removal if the track is inactive (age parameter), thus only the active tracks will be stayed.

Generally, the Deep SORT algorithm has a much better tracking performance and is more reliable than conventional methods. It is therefore an efficient tool in real-time situations where it is important to keep track of the same object over several frames, like the areas of surveillance, autonomous systems, and human-computer interaction.

5.2.3.3 BoT-SORT-ReID

BoTSORT (Bag of Tricks SORT) [75] is an advanced multi-object tracking (MOT) algorithm that improves tracking stability and accuracy by addressing common issues in dynamic environments. It is based on the SORT framework, and it takes up some of the significant features like Camera Motion Compensation (CMC), Kalman Filter (KF) state vector changes, and appearance-based matching (ReID) to carry out the tasks of occlusion and camera jitter.

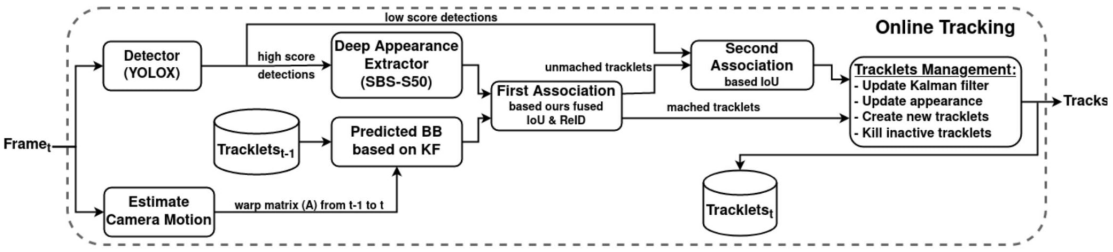


Figure 5.9: BoT-SORT-ReID tracker pipeline [76]

Key Features of BoTSORT

- *Camera Motion Compensation (CMC):* BoTSORT is the one that adopts Global Motion Compensation (GMC) that is carried out through the implementation of affine transformations to facilitate the removal of motion caused by the camera and consequently a more stable tracking. This is important for the outdoor tracking cases as the environmental conditions (e.g., wind or rain), may cause the camera to vibrate which in turn will interfere with the localization of the objects.

- *Kalman Filter State Vector Redesign:* The Kalman Filter (KF) that BoTSORT uses has been altered in such a way that it provides more accurate prediction performance. BoTSORT changes the state vector so that it tries to predict the width and height of the bounding box rather than the ratio of the dimensions, which are the parts used in methods such as DeepSORT and SORT. This innovation facilitates the localization during the object tracking.

- *IoU + ReID Fusion:* BoTSORT integrates Intersection over Union (IoU) along with ReID (Re-identification) techniques to improve the data association. The fusion of motion-based tracking as well as appearance features makes the algorithm able to augment the object matching to even the cases of the partial occlusion or changing in the appearance. The IoU-ReID fusion gives the guarantee that the most probable matches will be there only, therefore the accuracy and consistency in the multi-object tracking are improved.

- *Track Management:* BoTSORT uses a track management strategy where new tracks are created for unmatched detections and tracks for objects that have been lost for a long time are removed. The Hungarian algorithm is used for data association between predicted tracks and detections, which ensures the most accurate assignments based on the appearance and position of objects.

Tracking Process

The tracking procedure starts with object detection, then the classification of detections into two categories of high and low confidence. The Kalman Filter estimates the object's state, and data association is carried out on the basis of spatial and appearance similarity. The algorithm is not only capable of adjusting for camera motion but also for occlusions, ensuring robust tracking of objects in dynamic environments.

BoTSORT significantly enhances tracking robustness by combining motion compensation, improved data association with ReID, and advanced tracking management strategies, making it a reliable choice for multi-object tracking in complex scenes.

5.3 Image Registration and Calibration Procedure

5.3.1 Introduction

The process of image registration and calibration is essential for accurately aligning images from different sensors, such as RGB and thermal cameras, in multi-modal detection systems. This process thus involves several major stages in order to make sure that images from different angles or sensor types can be fused without any problem. In the context of this thesis, a camera calibration is done using a checkerboard pattern to get the intrinsic parameters such as the distortion coefficients. Calibration is followed by an image registration step

through the calculation of a homography matrix that enables a geometric transformation to be made between images from different cameras. The image alignment is performed using the homography matrix, which maps the corresponding points between the two camera modalities. Additionally, a cropping step is applied to ensure that both images have the same area of interest, allowing for seamless integration in multi-modal object detection. This section describes in depth the methods and the techniques used to perform cameras calibration and image registration in order to ensure that object detection and subsequent analysis will be done on the properly aligned image data.

5.3.2 Camera Calibration with Checkerboard

Camera calibration significantly influences image registration accuracy and take utmost importance when employing multi-modal sensor systems, such as RGB and thermal cameras. The camera calibration has been done in this thesis using a *checkerboard pattern* for each camera to find the distortion coefficients. This step is very important as it removes lens distortions and therefore, it facilitates more accurate registration between the images of different modalities which are by nature different from each other.

The values for the camera parameters used in the calibration are intentionally chosen to create *large differences* in the camera orientations (pitch, roll, and yaw) between the two modalities (RGB and IR). These big changes are not accidental but purposely created to represent very hard cases of alignment. In real-world applications, the alignment process would be much easier when the cameras have *closer values* for pitch, roll and yaw. In most practical applications, camera orientations (pitch, roll, yaw) are generally much closer to each other. Therefore, if the system performs well under large misalignments, it will undoubtedly handle *smaller differences* effectively, ensuring reliable performance when the camera setups are more aligned, as is often the case in real-world scenarios.

Table 5.1: Camera parameters for RGB and Thermal/IR cameras

Camera	FOV (degrees)	Width (pixels)	Height (pixels)	Pitch (degrees)	Roll (degrees)	Yaw (degrees)
RGB Camera	72	1920	1080	-35	-1	-2
IR Camera	57	640	512	-33	1	2

Pitch refers to the *up and down tilt* of the camera. A negative pitch means the camera is tilted downward, while a positive pitch means it's tilted upward. *Roll* refers to the *tilt around the forward axis* of the camera. A roll of 0 means the camera is perfectly level, while positive or negative values indicate tilts along the left-to-right axis. *Yaw* refers to the *left and right turning* of the camera around the vertical axis. A yaw of 0 indicates no turning, while positive or negative values indicate the camera is rotated to the right or left, respectively.

These numbers in Table 5.1 represent the situation where an *IR camera* usually has a *lower resolution* and a *smaller FOV (Field of View)* than the RGB camera. Such differences are considered during the alignment and registration process, thus the system is still robust enough to operate with these real-world differences between the two modalities.

A *12 pairs of images* set has been used for this calibration, where each pair consists of one image in the RGB modality and one in the thermal modality. In order to cover a vast amount of cases, these images have been taken from different *positions* and *angles* with respect to the checkerboard. By taking many pictures from different angles, the calibration becomes more precise as it can correct for changes in angle and distance that could lead to distortion in the images.

The thermal modality calibration is done by simulating that the white squares of the checkerboard are ”hot” and the black ones are ”cold”. The physical aspect of such experiment implies that a hot square is made of a material that reflects thermal energy, while the cold one is neutral in temperature. However, when it comes to the simulation, we are just approximating the heat distribution over the whole checkerboard. An example pair of images is shown in Figures 5.10 and 5.11, which illustrate the RGB and thermal images of the checkerboard pattern from different positions.

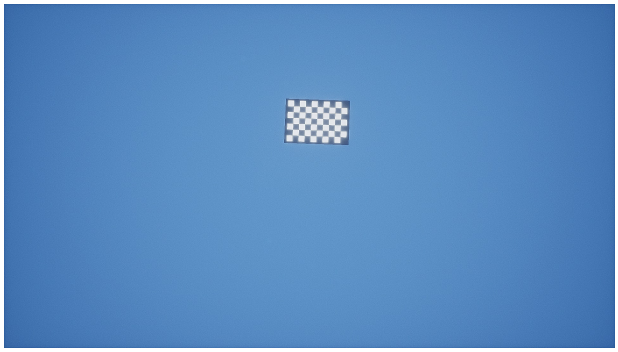


Figure 5.10: RGB checkerboard example

Figure 5.11: IR checkerboard example

The key purpose of this calibration is to find out the *distortion coefficients* for the camera lenses. The intrinsic matrix (for example, focal length and principal point) is a vital part of the full camera calibration, however, this process is done only to get the distortion coefficients. These coefficients correct the lens distortions present in the captured images, allowing for accurate image alignment in subsequent image registration steps.

5.3.3 Homography and Image Alignment

The homography results from the corresponding points in the pairs of images being the internal corners of the checkerboard in each modality. These corresponding corners are then matched between the two modalities. The internal corners of both modalities are shown in Figures 5.12 and 5.13.

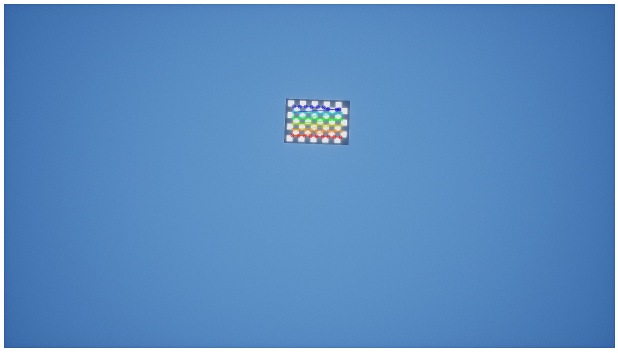


Figure 5.12: RGB checkerboard corners example

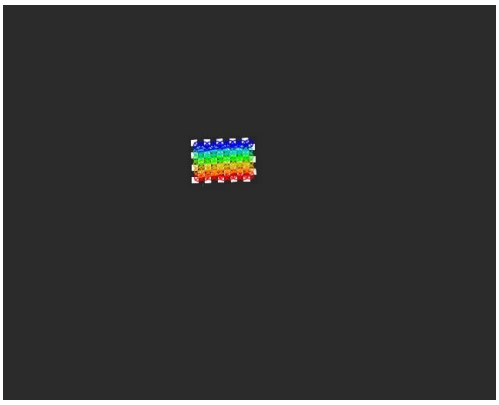


Figure 5.13: IR checkerboard corners example

In this process, the *RGB image* is considered the *reference modality*, meaning the coordinates and image layout from the RGB image will serve as the baseline for the alignment. The *IR* image is the *source modality*, which will be aligned with the RGB image.

The homography matrix is computed by matching corresponding points from the checkerboard pattern in both images. Using the identified matching points, the transformation from the IR image to the RGB image is calculated. This transformation allows for the alignment of the IR image to the RGB image, ensuring that the two modalities are geometrically aligned based on the matching points in both images.

Thus, the alignment process proceeds by transforming the coordinates of the IR image to fit the RGB reference, ensuring that both images share the same geometric space for subse-

quent analysis, such as multi-modal object detection. The final homography matrix, calculated from the internal corners of all pairs in the checkerboard across both modalities, is as follows:

$$H = \begin{bmatrix} 2.18366818 \times 10^{0} & -6.23326130 \times 10^{-2} & 3.73321724 \times 10^{2} \\ 1.29810614 \times 10^{-1} & 2.30138564 \times 10^{0} & -1.29773343 \times 10^{2} \\ -6.67200572 \times 10^{-5} & 1.02808865 \times 10^{-4} & 1.00000000 \times 10^{0} \end{bmatrix}$$

After computing the *homography matrix* that aligns the two images, the next step is to *undistort* the images. This process corrects any lens distortions present in the camera images, ensuring that the visual information is more accurate for further analysis.

Once the images are undistorted, the infrared image(*640x512*) is *aligned* with the RGB image(*1920x1080*) using the homography matrix. This alignment ensures that both images are correctly positioned relative to each other(final warped IR image *1920x1080*).

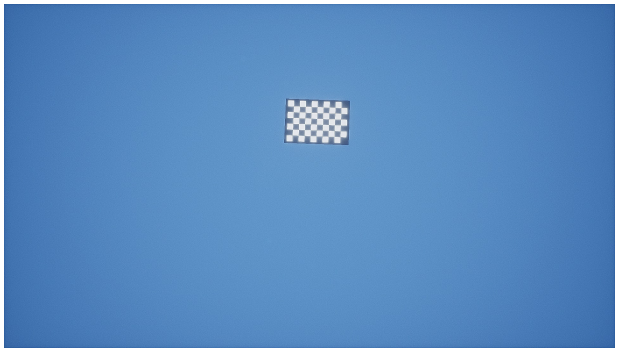


Figure 5.14: RGB original image

Figure 5.15: IR original image

Figure 5.16: Warped IR image

Next, the *regions of interest* in the aligned infrared image are isolated. This is done by converting the image to grayscale, applying a threshold to identify non-black areas (which correspond to the objects or features of interest), and then finding the contours of those areas.

After the contours are identified, the images are *cropped* to focus on the area containing the objects, with some margin added for flexibility. The cropping process also adjusts the aspect ratio of the images to match the target resolution of *640x512* with an aspect ratio of *1.25*. If the bounding box is too wide, the width is reduced to fit the desired aspect ratio; if the bounding box is too tall, the height is adjusted. The size of the cropped images before resizing is *1350x1080*, and this cropping process is done dynamically based on the contours.

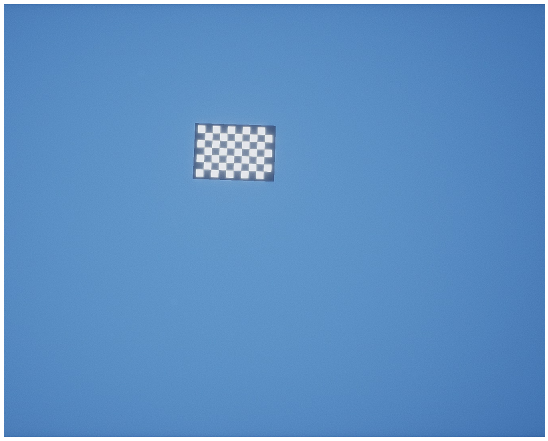


Figure 5.17: Cropped RGB image

Figure 5.18: Cropped IR image

Once the bounding box is adjusted to match the desired aspect ratio, both the infrared and RGB images are cropped to *1350x1080* and resized to *640x512* to ensure they are ready for seamless integration in multi-modal object detection.

Figure 5.19: Final resized real RGB image

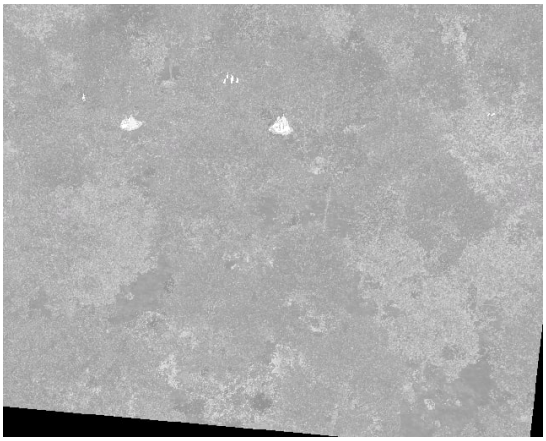


Figure 5.20: Final resized real IR image

5.4 ROS 2 Node Architecture

5.4.1 Introduction

The ROS 2 node architecture is the main factor that makes the system the most efficient, the nodes coordinate the various components and ensure that the operation of the system is done in an energy-efficient way. Each of them is built for one specific area of operation, which can be sensor data collection, image processing, object detection and tracking, and flight control.

Setting up the ROS 2 environment properly is the first necessary step to performing a successful execution of the system and that is discussed in the ROS 2 Setup section before going into the system's architecture and node descriptions. The environment setup should be done correctly, and all the dependencies must be met and the nodes must be set correctly for operation to be able to go ahead with the system execution effectively.

The current section illustrates the v that includes the custom ROS 2 message definitions which are the communication means for the nodes. ROS 2 package organization is also covered, highlighting the modular design.

The last part of this section analyzes the ROS 2 nodes and gives an insight into system launch and execution by illustrating the process of nodes' configuration and coordination.

5.4.2 ROS 2 Architecture Overview

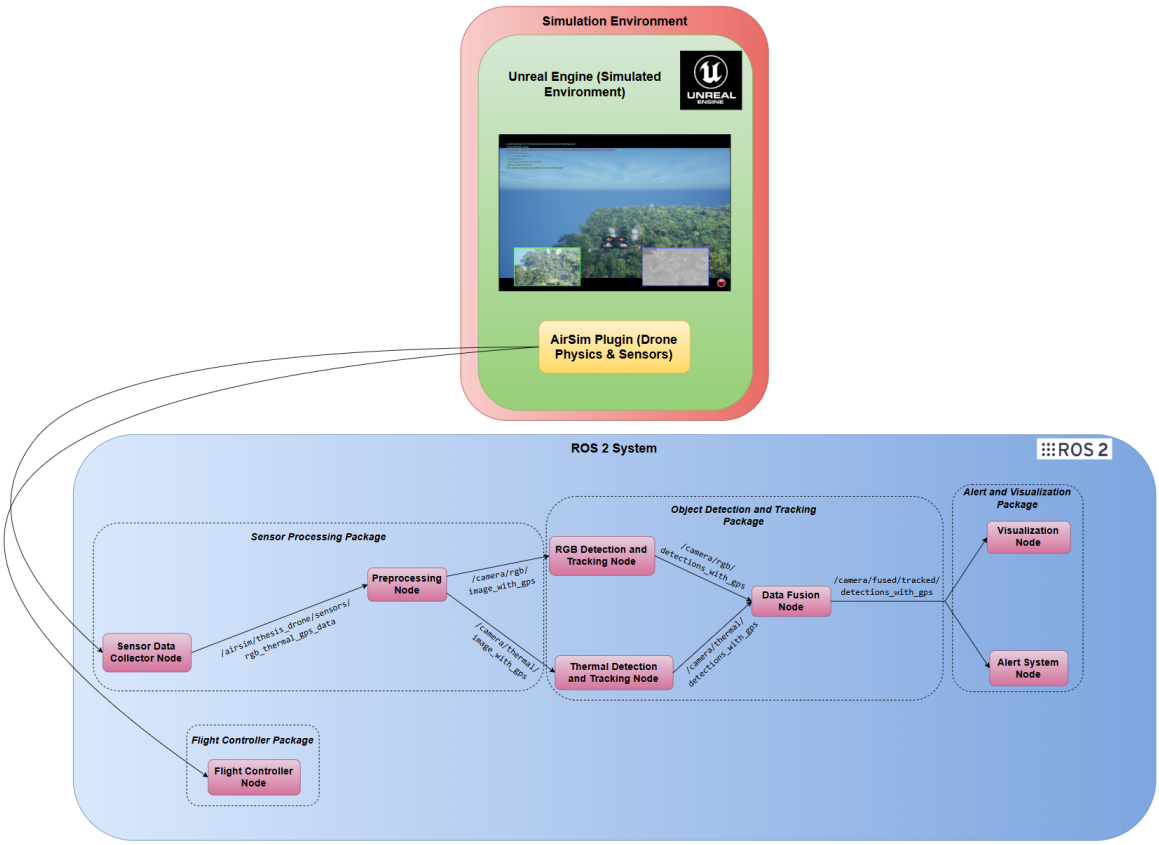


Figure 5.21: ROS 2 architecture

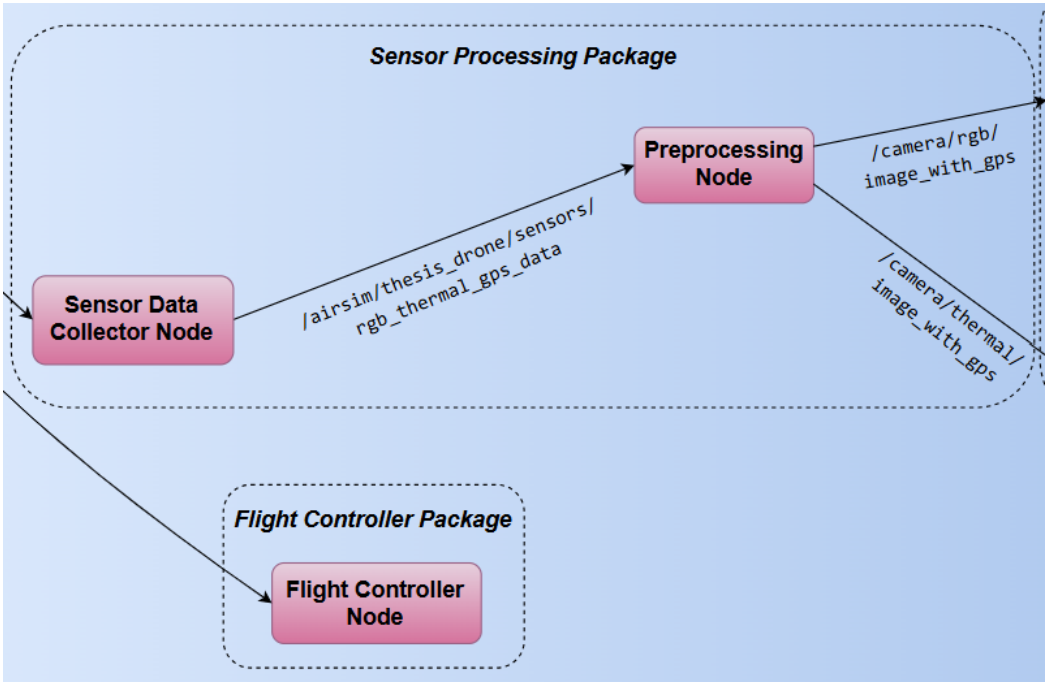


Figure 5.22: ROS 2 architecture part 1

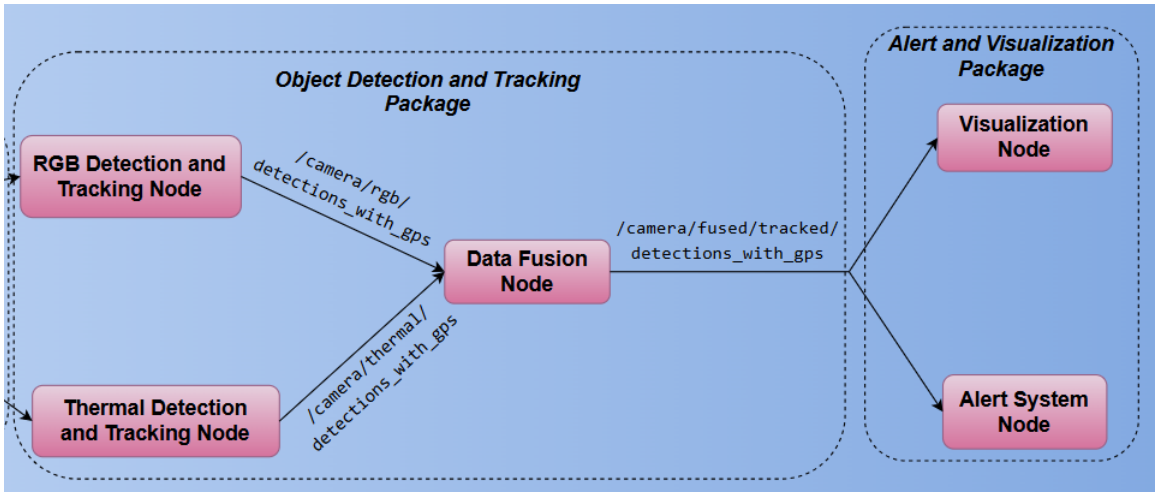


Figure 5.23: ROS 2 architecture part 2

As shown in Figure 5.21 the architecture integrates a simulation environment with a modular ROS 2 system, enabling real-time UAV-based detection of fire, smoke, humans, and animals using both RGB and thermal cameras. The simulation environment, that is analyzed in Chapter 6, is developed in *Unreal Engine*, with UAV dynamics and sensor simulation handled by the *AirSim plugin*, which provides realistic flight behavior and synchronized sensor data.

Within the ROS 2 system, the architecture is organized into distinct *packages*, each de-

signed for a specific operational task:

- *Sensor Processing Package:* This package includes nodes responsible for acquiring synchronized sensor data (RGB, thermal, and GPS) from AirSim and for aligning thermal images to the RGB frame using homography. The result is a consistent frame of reference for downstream detection tasks.
- *Object Detection and Tracking Package:* This package leverages deep learning models to perform object detection on both RGB and thermal modalities. It detects relevant targets such as fire, smoke, humans, and animals, tracks these objects using algorithms like DeepSORT, BoT-SORT, or ByteTrack, and associates the detection outputs with GPS metadata. The *Data Fusion Node* within this package combines the RGB and thermal detections into unified outputs, resolving overlaps and ensuring robust multi-modal tracking.
- *Flight Controller Package:* This package governs UAV movement by executing predefined GPS waypoint missions. It controls takeoff, rotation, and traversal patterns (e.g., zigzag coverage) to systematically monitor the environment.
- *Alert and Visualization Package:* This package is responsible for visualizing detections and issuing alerts. The visualization node renders bounding boxes, labels, confidence scores, and GPS positions on image streams using OpenCV. The alert system node monitors fused detections and logs alerts for fire, smoke, humans, and animals, including confirmation checks for fire using multi-modal detection consistency.
- *Msgs Package:* This package defines the custom ROS 2 message types used for communication between nodes. These message definitions enable seamless data exchange across the system and are described in detail in Section 5.4.3.

Communication between nodes is facilitated using *custom ROS 2 message definitions*, and the modular design ensures flexibility and scalability of the system. Detailed descriptions of each node are provided in the subsequent subsections.

5.4.3 Custom ROS 2 Message Definitions

To support modularity and structured data exchange between nodes, a dedicated `msgs` package was developed. This package defines all custom ROS 2 message types required for

transmitting synchronized sensor data, detection results, and fused outputs throughout the system. Below is a detailed overview of the defined messages:

- *SensorData.msg*
  This message encapsulates the raw synchronized data retrieved from the AirSim simulation environment. It includes:
  - `sensor_msgs/Image rgb_image`: the RGB frame from the drone's RGB camera.
  - `sensor_msgs/Image thermal_image`: the thermal frame from the thermal camera.
  - `sensor_msgs/NavSatFix gps`: the GPS coordinates at the time of image capture.

  This message is published on the topic:
  - `/airsim/thesis_drone/sensors/rgb_thermal_gps_data` by the *Sensor Data Collector Node*.

- *ImageWithGPS.msg*
  A simplified structure used for transmitting individual images with corresponding GPS data. It includes:
  - `sensor_msgs/Image image`
  - `sensor_msgs/NavSatFix gps`

  This message is published on the topics:
  - `/camera/rgb/image_with_gps` by the *Preprocessing Node*.
  - `/camera/thermal/image_with_gps` by the *Preprocessing Node*.

- *DetectionWithGPS.msg*
  This message type conveys the results of object detection and tracking, along with geolocation metadata. It contains:

  - `std_msgs/Header header`: standard ROS 2 header including timestamp and frame ID for temporal and spatial reference.
  - `string[] labels`: the predicted class labels (e.g., fire, human).
  - `float32[] confidences`: detection confidence scores for each label.
  - `float32[] boxes`: bounding box coordinates, formatted as `[x1, y1, x2, y2, ...]`.
  - `int32[] tracking_ids`: unique IDs for tracked objects across frames.
  - `sensor_msgs/NavSatFix gps`: GPS location associated with the detection.
  - `sensor_msgs/Image image`: image frame on which detections are visualized.

  This message is published on the topics:

  - `/camera/rgb/detections_with_gps` by the *RGB Detection and Tracking Node*.
  - `/camera/thermal/detections_with_gps` by the *Thermal Detection and Tracking Node*.

- *TrackedObject.msg*

  Represents a single detected and tracked object. Its fields include:

  - `int32 id`: the unique identifier for the tracked object.
  - `string label`: class label assigned to the object.
  - `float32 confidence`: model confidence in the detection.
  - `float32[4] box`: bounding box defined by the coordinates `[x1, y1, x2, y2]`.
  - `string source`: identifies the source modality of the detection (e.g., `"rgb"` or `"thermal"`).

- *TrackedDetections.msg*

  Combines all tracked detection data from the RGB and thermal pipelines after fusion. It includes:

  - `sensor_msgs/NavSatFix gps`: global location of the fused detections.
  - `TrackedObject[] tracks`: list of tracked object instances.
  - `sensor_msgs/Image rgb_image`: corresponding RGB image frame.
  - `sensor_msgs/Image thermal_image`: corresponding thermal image frame.

  This message is published on the topic:

  - `/camera/fused/tracked/detections_with_gps` by the *Data Fusion Node*.

The following are brief descriptions of commonly used standard ROS 2 message types:

- *sensor_msgs/Image:*

  Represents an image frame as a matrix of pixel values. It includes metadata such as height, width, encoding (e.g., `rgb8` or `mono8`), endianness, step size, and the actual image data array.

- *sensor_msgs/NavSatFix:*

  Contains geographical information including latitude, longitude, altitude, position covariance and status indicators. It is compliant with GNSS specifications (Global Navigation Satellite System) and is used to geolocate sensor data.

- *std_msgs/Header:*

  A standard message header used to timestamp messages.

These custom message types, along with their standard dependencies, ensure type-safe, structured communication across all ROS 2 nodes and provide a clean abstraction layer for interacting with image, detection, and geolocation data.

5.4.4 Node Descriptions

5.4.4.1 Sensor Data Collector Node

The Sensor Data Collector Node is responsible for acquiring synchronized RGB, thermal, and GPS data from the simulated UAV in the Unreal Engine environment via the AirSim API. This node establishes a persistent connection with the AirSim simulator, enabling API control and arming the UAV. At a fixed rate of 10 FPS, the node captures images from two onboard cameras: an RGB camera and a custom thermal camera labeled as `FireVisionThermal`. The selected frame rate of 10 FPS strikes an optimal balance between computational efficiency and temporal resolution, making it well-suited for UAV-based wildfire detection. In such applications, the environment typically evolves slowly, fire propagation or human presence does not change significantly within a single second, thus higher frame rates offer limited practical benefit while increasing computational overhead. Therefore, 10 FPS ensures reliable detection and consistent data fusion without unnecessary resource consumption. The retrieved image frames are converted to ROS-compatible messages using OpenCV and `cv_bridge`, ensuring proper encoding for downstream nodes. Simultaneously, GPS data including latitude, longitude, and altitude is retrieved and encapsulated in a `NavSatFix` message. All three sensor streams, RGB image, thermal image, and GPS coordinates, are bundled into a single `SensorData` message and published on the topic `/airsim/thesis_drone/sensors/rgb_thermal_gps_data`. This node thus serves as the foundational data provider for the rest of the system, ensuring temporally aligned and spatially meaningful input to all subsequent modules.

5.4.4.2 Preprocessing Node

The Preprocessing Node is responsible for aligning and formatting the raw sensor data collected by the UAV into a consistent spatial and temporal format, preparing it for downstream object detection and tracking. It subscribes to the `/airsim/thesis_drone/sensors/rgb_thermal_gps_data` topic, where raw RGB and thermal images, along with GPS coordinates, are published as part of the `SensorData` message. The core objective of this node is to spatially align the thermal image with the RGB frame, crop the overlapping region, and standardize the output resolution for both image modalities.

Upon initialization, the node loads a precomputed homography matrix from a configu-

ration file (`homography_matrix.npz`) using NumPy. This matrix was previously estimated through a checkerboard-based calibration procedure, as explained in Subsection 5.3.2. The homography is used to geometrically transform the thermal image so that it aligns with the coordinate space of the RGB image. This transformation is applied using OpenCV's `warpPerspective` function, enabling the projection of the thermal image onto the RGB plane. Following this alignment, the overlapping region is identified and cropped, and both image modalities are resized to a standardized resolution of 640×512, as detailed in Subsection 5.3.3, ensuring consistent input dimensions for downstream processing.

The RGB and aligned thermal images are repackaged with GPS metadata into `ImageWithGPS` messages. These are published on the topics:

- `/camera/rgb/image_with_gps`
- `/camera/thermal/image_with_gps`

This preprocessing step is crucial, as it guarantees that both RGB and thermal streams are not only temporally synchronized but also spatially aligned. The homography-based alignment, which is static throughout the mission, ensures that corresponding objects appear in the same relative positions across both modalities, enabling effective multi-modal fusion in later stages of the pipeline. However, because this transformation depends on the parameters of the cameras, any modification to their configuration (e.g., position, orientation, or lens properties) would invalidate the existing homography matrix. In such cases, the registration process must be repeated, as described in Section 5.3, to generate an updated and valid transformation.

Additionally, preserving GPS metadata ensures that each image can be geolocated, which is critical for alert generation, environmental mapping, and fire response applications. Overall, this node encapsulates both the image registration and formatting logic required to prepare real-world, multi-modal UAV data for accurate and efficient detection. The combination of image warping, contour-based cropping, resolution normalization, and metadata preservation ensures a high level of consistency and robustness in the visual perception pipeline.

5.4.4.3 Object Detection and Tracking Nodes

The system includes two detection and tracking nodes, each dedicated to a specific visual modality. These nodes are structurally similar, sharing a common architecture for perform-

ing real-time object detection with YOLOv8 and tracking with a configurable multi-object tracker. The primary differences lie in the input source topic, model selection, and class labels of interest.

Both nodes subscribe to `ImageWithGPS` messages, which contain preprocessed image frames and GPS metadata. The RGB node listens on the `/camera/rgb/image_with_gps` topic, while the thermal node listens on `/camera/thermal/image_with_gps`. Upon receiving a message, each node converts the ROS image message into an OpenCV-compatible format using `cv_bridge`. Detection is then performed using a pre-trained *YOLOv8s* model optimized with *TensorRT in FP16 precision*, ensuring high inference speed with reduced resource usage, especially important for deployment on edge devices such as Jetson platforms.

Each node loads the correct model dynamically from the shared ROS package `object_detection_and_tracking_pkg` in the initialization of the node. Specifically:

- The *RGB Detection and Tracking Node* loads the model from `weights/yolo_rgb.engine`.
- The *Thermal Detection and Tracking Node* loads the model from `weights/yolo_thermal.engine`.

Detection is followed by object tracking, using one of the supported algorithms: *DeepSORT*, *BoT-SORT-ReID*, or *ByteTrack*. Only one tracker is used at a time, selected dynamically at runtime via the ROS 2 parameter *tracker_type*.

The *BoT-SORT-ReID* tracker, like DeepSORT, typically uses feature maps from the backbone of the detection model (such as YOLOv8s) for ReID (Re-Identification). However, since YOLOv8s is optimized with TensorRT, direct access to the model's backbone is not possible, limiting the ability to extract feature maps. To address this, a new classification model is used to extract the feature maps for ReID, with the model optimized in TensorRT using INT8 precision to ensure fast, efficient processing. In future work, the goal is to modify TensorRT optimization to provide both detection results and feature maps, enabling full access to the necessary data for tracking and object association while maintaining high performance in real-time applications.

The configuration for the BoT-SORT tracker includes the following key settings:

- *tracker_type = botsort:* The chosen tracker type.

- *track_high_thresh = 0.4:* The threshold for the first association (initial matching).
- *track_low_thresh = 0.3:* The threshold for the second association (refinement step).
- *new_track_thresh = 0.4:* The threshold for initializing new tracks when detections do not match existing tracks.
- *track_buffer = 50:* The buffer size to calculate when to remove tracks based on time.
- *match_thresh = 0.8:* The threshold for matching tracks based on confidence.
- *fuse_score = True:* Whether to fuse confidence scores with the IoU distances before matching.
- *gmc_method = sparseOptFlow:* Method for global motion compensation, crucial for handling large shifts in object movement.
- *proximity_thresh = 0.5:* The minimum IoU threshold for valid matches with ReID.
- *appearance_thresh = 0.8:* The minimum appearance similarity for ReID.
- *with_reid = True:* Whether to include ReID for more robust tracking across frames.
- *model = "yolov8n-cls.engine":* Uses the TensorRT YOLOv8n classification model in INT8 precision.

The *ByteTrack*, like BoT-SORT, offers another solution for multi-object tracking, providing flexibility when combined with YOLO-based models. It uses similar configurations for track association and matching, but has optimizations suitable for handling high-speed detections with a focus on reducing false positives and improving tracking accuracy in real-world scenarios. The ByteTrack does not have configurations about GMC and ReID.

For both trackers, the node dynamically loads the configuration and tracks objects using the respective parameters provided in the launch file.

For *DeepSORT*, the nodes instantiate the tracker with the following parameters:

- *max_age = 50:* Maximum number of missed detections before a track is deleted.
- *max_iou_distance = 0.5:* IoU threshold for associating detections with existing tracks.
- *nms_max_overlap = 0.5:* Non-Maximum Suppression threshold.

- *embedder = "mobilenet":* Lightweight embedding model used for appearance feature extraction.
- *n_init = 3:* Minimum number of detections before a track is confirmed.

Once predictions are obtained, the results are packaged into a `DetectionWithGPS` message as explained in Subsection 5.4.3 and are published on:

- `/camera/rgb/detections_with_gps` for the RGB node.
- `/camera/thermal/detections_with_gps` for the thermal node.

If no detections are received for 50 consecutive frames, the node resets the internal state of the tracker to prevent uncontrolled growth of track IDs. This mechanism ensures that the tracker remains synchronized with the scene, maintaining ID continuity and avoiding drift in cases of instantaneous optical degradation, temporary occlusion, or poor visibility conditions.

Together, these two detection nodes form the core of the perception subsystem, transforming visual inputs into geolocated semantically meaningful detections with persistent identity tracking across time.

5.4.4.4 Data Fusion Node

The Data Fusion Node is responsible for integrating detection results from the RGB and thermal modalities into a unified representation. It plays a critical role in combining complementary information from both sensor streams, enabling more robust and reliable perception in multimodal UAV systems. This node subscribes to two input topics:

- `/camera/rgb/detections_with_gps` – containing RGB-based detection and tracking results,
- `/camera/thermal/detections_with_gps` – containing thermal-based detection and tracking results.

The node utilizes an `ApproximateTimeSynchronizer` that matches RGB and thermal messages from two asynchronous data streams, based on the closest timestamps, to achieve synchronization. The `slop` parameter is configured to 0.1 seconds, which permits minor time differences between sensor modalities and still ensures that the valid matches are not discarded. Once a pair of synchronized messages is received, the fusion process is executed via a callback function.

Within the fusion callback, a new message of type `TrackedDetections` is constructed as explained in Subsection 5.4.3.

Each individual detection from the RGB and thermal messages is transformed into a `TrackedObject` as explained in Subsection 5.4.3.

The combined list of `TrackedObject` instances (to differentiate between modalities, labels for fire detections are modified to *rgb_fire* and *thermal_fire*) is assigned to the `tracks` field of the `TrackedDetections` message, which is then published on the topic `/camera/fused/tracked/detections_with_gps` for consumption by higher-level modules such as the alert or visualization system.

This fusion method retains the information specific to each modality and at the same time gives a single view of the detected area. Even though the fusion is done as a simple concatenation of detection results obtained from different modalities without a geometric merge or an IoU-based association, it still efficiently converts the data into a form that is suitable for subsequent decision-making by keeping the traceability through the `source` field and the track IDs.

To sum up, the Data Fusion Node guarantees both the time synchronization and the semantic combination of the detection results obtained from multiple sensors, thus enabling the creation of a reliable situational awareness in UAV-based wildfire monitoring applications.

5.4.4.5 Flight Controller Node

The Flight Controller Node is responsible for autonomously navigating the UAV within the AirSim simulation environment using a predefined set of GPS-based waypoints. Upon startup, the node connects to AirSim, enables API control, and arms the UAV. The drone takes off from the initial position defined by the `Player Start` object in the Unreal Engine environment.

The navigation pattern is based on a dense sequence of GPS coordinates forming a zigzag trajectory to ensure full coverage of the surveillance area. At each waypoint, the UAV rotates toward the target direction before moving forward, maintaining consistent orientation during flight. The waypoint path, as shown in Figure 5.24, illustrates the planned coverage over the mapped terrain.

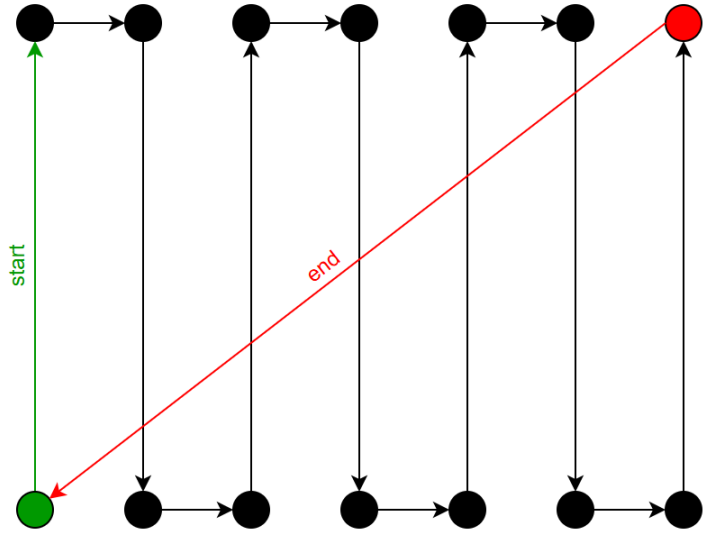


Figure 5.24: Flight path followed by the UAV

Importantly, the ROS 2-based implementation enables parallel execution across all nodes. The Flight Controller Node operates independently of other nodes, ensuring that UAV motion and navigation do not interfere with the image processing or data publishing pipelines. This parallelism enhances system modularity and scalability, allowing seamless real-time operation of multiple components.

The flight sequence operates in a loop, allowing continuous patrolling. Safe shutdown procedures are in place to disarm the UAV and disable API control when the node is terminated. This flight behavior supports the system's goal of repeatable data acquisition for real-time object detection and tracking.

5.4.4.6 Alert System Node

The Alert System Node is responsible for monitoring fused detection results and generating alerts for critical events such as fire, smoke, human, or animal presence. It subscribes to the topic `/camera/fused/tracked/detections_with_gps`, which carries the output of the Data Fusion Node in the form of `TrackedDetections` messages. Each message includes a list of tracked objects from both RGB and thermal modalities, along with corresponding GPS coordinates and raw images.

Upon receiving a message, the node evaluates all incoming detections and filters them based on both class label and modality. To manage alert frequency and avoid duplicates, the system maintains two separate sets of seen track IDs: one for RGB-based detections and one

for thermal-based detections. Each ID set is periodically reset if no detections are received for 50 consecutive frames, which ensures that long-term occlusions or missed detections do not permanently suppress alert generation.

Fire detection is handled with a confidence-aware logic. If a fire is detected in both modalities and the bounding boxes have an Intersection over Union (IoU) score of at least 0.5, a combined alert is triggered. In cases where fire is detected only in one modality, alerts are still issued if the confidence exceeds a threshold: 0.7 for RGB-only detections and 0.5 for thermal-only detections. This conservative yet flexible rule ensures robust fire confirmation while avoiding false positives from weak signals.

In addition to fire, the node also monitors for smoke (from RGB) and for humans or animals (from thermal). Each new detection with a previously unseen ID is logged, and if the object is of type smoke, it is also flagged as a high-priority warning. All alerts include the object type, detection confidence, unique ID, and the current GPS coordinates of the drone.

The node prints all warning messages to the terminal with timestamps, allowing human real-time monitoring or redirection to the logging systems. A sample log entry may include GPS coordinates and multiple alert lines, e.g. combined fire detection, smoke presence, or thermal signatures of humans or animals.

This node operates independently and continuously within the ROS 2 framework, enabling real-time alert generation without interfering with other system components. Its modularity and decision logic make it a critical component for field applications where timely and reliable notifications are required for disaster response or environmental monitoring.

5.4.4.7 Visualization Node

The Visualization Node is responsible for visualizing the tracked detections on both RGB and thermal images and displaying them in real-time. This node subscribes to the topic `/camera/fused/tracked/detections_with_gps`, where the fused detection results are published. These results contain information such as detection labels, bounding boxes, confidence scores, tracking IDs, and GPS metadata.

Upon receiving a `TrackedDetections` message, the node attempts to convert the provided RGB and thermal image frames into OpenCV-compatible formats using `cv_bridge`. The node then processes the detections by iterating through the detected objects. For each detection, it assigns appropriate colors and labels based on the detection type and draws the

bounding boxes on the images. Additionally, each detected object is annotated with a tracking ID and confidence score, which is displayed as text near the bounding box.

The node also overlays the GPS data (latitude, longitude, and altitude) on the images to provide real-time geospatial context for the detections. The visualization is updated continuously for both RGB and thermal frames, with separate windows displayed for each modality using OpenCV.

This node runs in real-time within the ROS 2 event loop and visualizes the data using OpenCV, however, it is worth mentioning that this method is very demanding in terms of computing power which is especially true if it is executed on a UAV or an embedded device. In the practical real-world situation, the images with the detection results are going to be sent to a ground station in order to be displayed and analyzed further. By doing so, the UAV can free up its resources to concentrate on more critical jobs like sensor data acquisition, object detection, tracking, and so on, while the ground station takes care of the resource-intensive task of rendering and displaying the images.

In summary, the Visualization Node serves as a crucial part of the system, enabling real-time display of tracking results on both RGB and thermal images. However, for practical implementation on a UAV, the visualization function would be offloaded to a ground station to mitigate the computational load on the drone's onboard systems.

5.4.5 System Setup and Execution

The system is launched using a ROS 2 launch file, `ros2_system_launch.py`, located in the `sensor_processing_pkg` package. This launch file coordinates the execution of various nodes that make up the system, ensuring they are properly initialized and configured for autonomous UAV operation.

The launch file begins by declaring the *tracker_type* argument, which allows the user to dynamically select the tracking algorithm for both the RGB and thermal detection nodes. The available options for tracker_type include *deepsort* (default), *botsort* and *bytetrack*.

Once the tracking algorithm is selected, the launch file proceeds to initialize the necessary nodes, including the *Flight Controller Node*, *Sensor Data Collector Node*, *Preprocessing Node*, *Object Detection and Tracking Nodes*, *Data Fusion Node*, and *Alert and Visualization Nodes*. These nodes work together to collect sensor data, perform object detection, track identified objects, and generate alerts based on fused detections.

To start the entire system, use the following ROS 2 command:

```
$ ros2 launch sensor_processing_pkg ros2_system_launch.py tracker_type:=
   deepsort
```

This command initiates the launch with the default deepsort tracker. If a different tracking algorithm is preferred, the tracker_type argument can be modified to select one of the available options. This setup provides flexibility in selecting dynamically the tracking algorithm for the given application, ensuring efficient and accurate object tracking and detection. But before this command if we open a new terminal we must run the following command to set the configurations:

```
$ source install/setup.bash
```

In case we change something in the packages or configurations, we must rebuild the system by running the following command:

```
$ colcon build --symlink-install
```

Chapter 6

Simulation Environment Implementation

6.1 Introduction

Simulation environments play a vital role in the experimentation and verification of UAV systems, especially for intricate jobs like fire detection. We outline a high-fidelity simulation environment built specifically for UAV wildfire detection and also the remaining classes from both modalities in this chapter. The basic intent is to set up a virtual place for the testing of different sensors, among them a thermal camera designed specifically, for the detection and tracking tasks in the UAV system.

In this regard, we deploy Unreal Engine, a fast real-time 3D engine, along with AirSim, a simulation platform that Microsoft designed for autonomous vehicles. Unreal Engine is visualizing the place where the UAV moves, while AirSim is helping the set-up of the various vehicle sensors and flight dynamics. The use of this mixture makes it possible for a realistic and interactive simulation that is able to simulate the real conditions.

Together, these components provide a robust simulation platform for evaluating UAV performance under realistic conditions, crucial for fine-tuning the wildfire detection before deployment in real-world scenarios.

This chapter details the process of creating the virtual environment and integrating it with AirSim to simulate UAV flight and sensor interactions and also describes the design and integration of a custom thermal camera into the simulation environment.

6.2 Setup for Simulation

6.2.1 Introduction

This section will explore the steps involved in creating the virtual environment, detailing both the setup of the terrain and assets, and the integration of AirSim with Unreal Engine to enable communication between the simulation and the UAV system. Through this process, we create a highly interactive and realistic virtual environment for testing and optimizing UAV-based wildfire detection systems.

6.2.2 AirSim Simulator

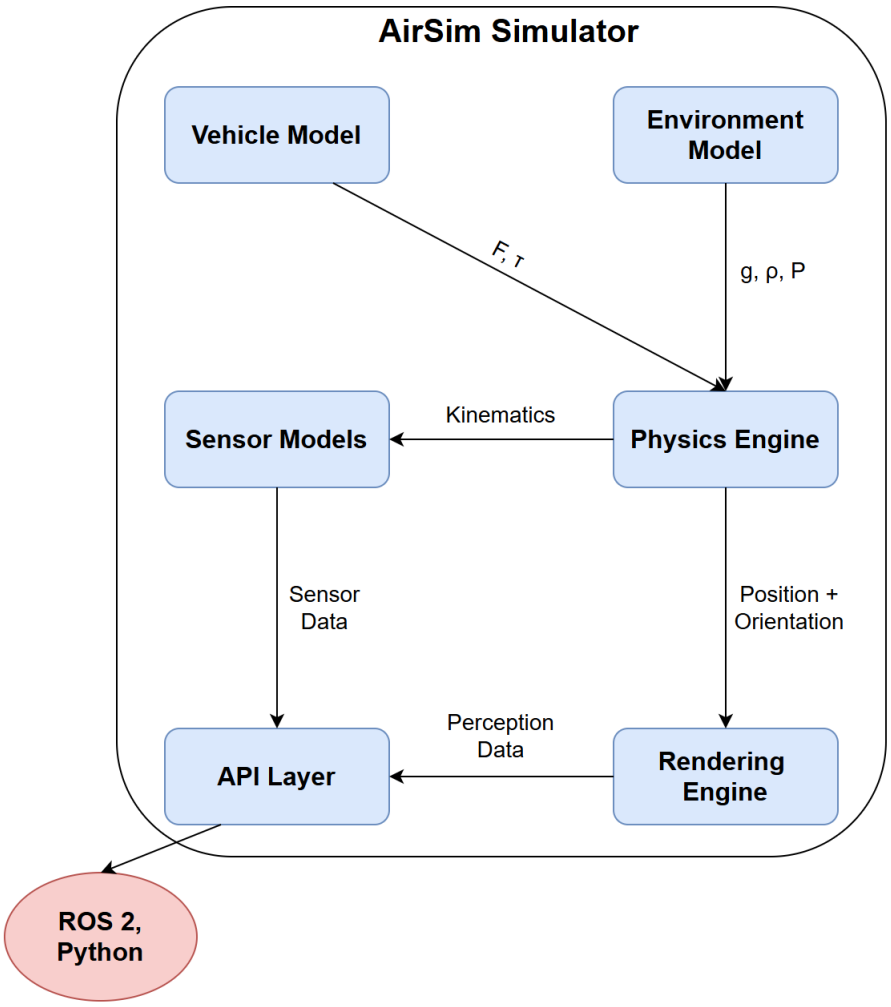


Figure 6.1: AirSim Internal Simulation Architecture

The diagram in Figure 6.1 illustrates the internal architecture of *AirSim* as a modular simulator, showing how different components interact to realistically simulate vehicle motion,

sensor feedback, and perception in real time. The *Vehicle Model* converts them into physical forces and torques (F, τ), reflecting how a real vehicle would respond. These quantities are sent to the *Physics Engine*, which uses environmental parameters such as gravity (g), air density (ρ), and terrain properties obtained from the *Environment Model* to compute the vehicle's motion. As a result, the Physics Engine updates both the *pose* (position and orientation) and *kinematics* (velocities, accelerations), which are propagated to other subsystems.

The updated pose is passed to the *Rendering Engine*, which leverages Unreal Engine to generate realistic visual outputs such as RGB or thermal images. These are returned to the API Layer as *perception data*, simulating what the onboard cameras would see. Simultaneously, *Sensor Models* (e.g., IMU, GPS, LiDAR) receive the kinematic state from the Physics Engine to simulate *sensor data* such as accelerations, angular velocities, or global coordinates. Both perception and sensor data flow through the API Layer to external applications (e.g., ROS, Python clients), enabling closed-loop control, training of machine learning models, and autonomous system testing. This architecture enables AirSim to simulate complex environments and physically accurate UAV behavior with high visual fidelity, making it a powerful tool for research and development in autonomous systems.

6.2.3 Virtual Environment Overview

Initially, we follow the instructions provided by the official Unreal Engine documentation for Linux installation. Since the Epic Games Launcher is not supported on Linux, we download Unreal Engine directly from the Unreal Engine Documentation.

After downloading, we navigate to the `Engine/Binaries/Linux` directory and execute the `UnrealEditor` binary file using the following command:

```
$ ./UnrealEditor
```

Subsequently, the projects are stored in the `"Unreal Projects"` folder, which is located within the `Documents` directory.

Additionally, we change the permissions for the Unreal Engine folder and its subdirectories to allow full access. This is done using the following commands:

```
$ sudo chmod -R 777 unreal
$ sudo chown -r username unreal
```

In cases where a project is created using a different version of Unreal Engine, it is necessary to manually rebuild the project after importing it into the editor. This ensures compatibility with the current engine version and prevents potential runtime issues. The following command can be used to build the project:

```
$ /home/ubuntu/unreal/Engine/Build/BatchFiles/Linux/Build.sh
-Development Linux -Project="myproject.uproject" -TargetType=Editor -
    Progress
-NoEngineChanges -NoHotReloadFromIDE
```

On Linux systems, Unreal Engine projects use a unique hash code instead of a human-readable version string for the `EngineAssociation` field. This field is located within the root of the project's `.uproject` file in JSON format, for example:

```
"EngineAssociation": "E5ECE0DB-D57F-4904-9E03-1E7581B3F5FC",
```

As a result, even if a project is downloaded from the Marketplace and is created with the same nominal version of Unreal Engine, it must still be compiled and rebuilt to match the exact Linux installation. This is because the Linux engine version is identified internally by its unique hash code, and Unreal Engine requires a match to proceed with project compilation and plugin compatibility.

For the creation of the forest, the *Electric Dreams Environment* from the FAB Marketplace was utilized. The environment can be accessed through the following link: Electric Dreams Environment on FAB Marketplace. An example of the forest generated in the simulation is shown in Figure 6.2.

Figure 6.2: Forest environment

In the forest, additional animals such as deer and wolves were added using the assets available from the FAB Marketplace. The specific assets used can be accessed via the following link: FAB Marketplace - Animals (Deer and Wolves). Humans were also added to the simulation using assets from Mixamo, which includes both character models and animations. The color of the models is not a concern, as both humans and animals are utilized exclusively for the thermal camera in the simulation.

Figure 6.3: Deer asset

Figure 6.4: Wolf asset

Figure 6.5: Human asset

For the creation of fire and smoke, the project from GitHub, Firefly 2.0, was utilized. This project includes various assets for both fire and smoke effects. Below are examples of the assets utilized in the environment.

Figure 6.6: Smoke asset

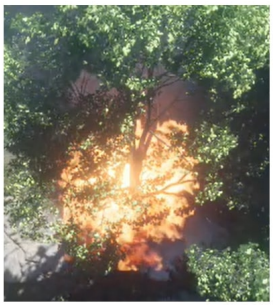


Figure 6.7: Fire asset

Figure 6.8: Fire and smoke assets

The final environment that will be used to run the simulation is shown in Figure 6.9. It includes all the assets for fire, smoke, humans, and animals distributed throughout the scene.

Figure 6.9: Final environment for simulation

6.2.4 AirSim Integration with Unreal Engine

The official AirSim project developed by Microsoft has been discontinued. As a result, two alternatives have emerged: the *Cosys-AirSim* fork developed by Cosys-Lab[1], and the *Colosseum* fork from CodexLabsLLC. However, the Colosseum branch currently lacks compatibility with Ubuntu 22.04 due to Vulkan-related issues. Therefore, in this thesis, we selected the Cosys-AirSim fork (specifically the `5.2.1` branch) as it provides stable support for both Unreal Engine 5 and Ubuntu 22.04.

The integration process involves the following steps:

[1] https://github.com/Cosys-Lab/Cosys-AirSim/tree/5.2.1

1. Clone the repository:

```
$ git clone -b 5.2.1 https://github.com/Cosys-Lab/Cosys-AirSim.
    git
```

2. Navigate into the AirSim directory:

```
$ cd Cosys-AirSim-5.2.1
```

3. Run the setup and build scripts:

```
$ ./setup.sh
$ ./build.sh
```

Once AirSim is built, navigate to the `Unreal` folder inside the repository, copy the `Plugins` directory, and paste it into the Unreal project directory (at the same level as the `myproject.uproject` file).

In the Unreal project, open the `Config/DefaultGame.ini` file and append the following lines:

```
+MapsToCook=(FilePath="/AirSim/AirSimAssets")
+DirectoriesToAlwaysCook=(Path="/AirSim/HUDAssets")
+DirectoriesToAlwaysCook=(Path="/AirSim/Beacons")
+DirectoriesToAlwaysCook=(Path="/AirSim/Blueprints")
+DirectoriesToAlwaysCook=(Path="/AirSim/Models")
+DirectoriesToAlwaysCook=(Path="/AirSim/Sensors")
+DirectoriesToAlwaysCook=(Path="/AirSim/StarterContent")
+DirectoriesToAlwaysCook=(Path="/AirSim/VehicleAdv")
+DirectoriesToAlwaysCook=(Path="/AirSim/Weather")
```

In the `Source` folder, inside the two C# scripts, ensure that the build version is specified as:

```
DefaultBuildSettings = BuildSettingsVersion.V2;
```

and not set to `V5`.

Additionally, update the `.uproject` file to include the AirSim plugin dependency:

```
"AdditionalDependencies": [
    "AirSim"
]
```

and enable the plugin as follows:

```
{
    "Name": "AirSim",
    "Enabled": true
}
```

To configure the drone's capabilities, modify the `settings.json` file located in the `Documents/AirSim` folder (created by default). Below is a partial `settings.json` configuration for RGB and infrared cameras used in this simulation:

```
{
  "SettingsVersion": 2.0,
  "SimMode": "Multirotor",
  "RpcEnabled": true,
  "Vehicles": {
    "SimpleFlight": {
      "VehicleType": "SimpleFlight",
      "VehicleName": "ThesisDrone",
      "AllowAPIAlways": true,
      "Cameras": {
        "my_rgb_camera": {
          "CaptureSettings": [
            {
              "ImageType": 0,
              "Width": 1920,
              "Height": 1080,
              ...
            }
          ],
          "X": 0, "Y": -0.3, "Z": 10,
          "Pitch": -35, "Roll": -1, "Yaw": -2
        },
        "my_infrared_camera": {
          "CaptureSettings": [
            {
              "ImageType": 10,
              "Width": 640,
              "Height": 512,
              ...
```

```
            }
          ],
          "X": 0, "Y": 0.3, "Z": 10,
          "Pitch": -33, "Roll": 1, "Yaw": 2
        }
      }
    }
  },
  "SubWindows": [
    {"WindowID": 0, "CameraName": "my_rgb_camera", "ImageType": 0, ...},
    {"WindowID": 2, "CameraName": "my_infrared_camera", "ImageType": 10,
      ...}
  ]
}
```

6.3 Custom Thermal Camera Creation

6.3.1 Introduction

Since there was no prepared infrared camera config available in Unreal Engine or AirSim, the decision was taken to develop a custom thermal camera. The information available in the documentation and manuals is quite sparse and, therefore, no appreciable depth for comprehensive integration are found there. Thus, this work takes on a project from GitHub, whose features and implementation details served as a valuable starting point. Nevertheless, in order to match the specific needs of this research, thermal appearance has to go through many changes in the process of its improvement. It is extremely important to be able to represent thermal in a realistic way since the main objective is to represent a UAV-based wildfire detection system which is equipped with both RGB and IR cameras. Accurate thermal visualization ensures that the forest, fire, humans, and animals are distinguishable within the simulated environment. Special attention is given to differentiating the forest's thermal appearance to simulate a more realistic thermal landscape, which is essential for the development and evaluation of reliable detection models.

6.3.2 Thermal Camera Integration with Unreal Engine

For the creation of the thermal camera, the *firevision_sim* project[2] on GitHub was highly helpful. However, significant modifications were made, primarily in the post-processing material, to develop a customized thermal camera effect. Two materials were used in sequence. The first material assigns temperature values across the scene and defines hot and cold regions based on the thermal properties of different objects. The second material enhances the visual realism by refining the thermal appearance, removing sun shadows, smokes and other elements that are not typically captured by an infrared camera. Examples of the thermal output after applying only the first material and after applying both materials can be seen in Figures 6.16 and 6.17, respectively, illustrating the specific role and applicability of each material in the simulation pipeline. The Figures 6.10, 6.11, 6.12 and 6.13, presented below, are connected to the Figure 6.14 at the end.

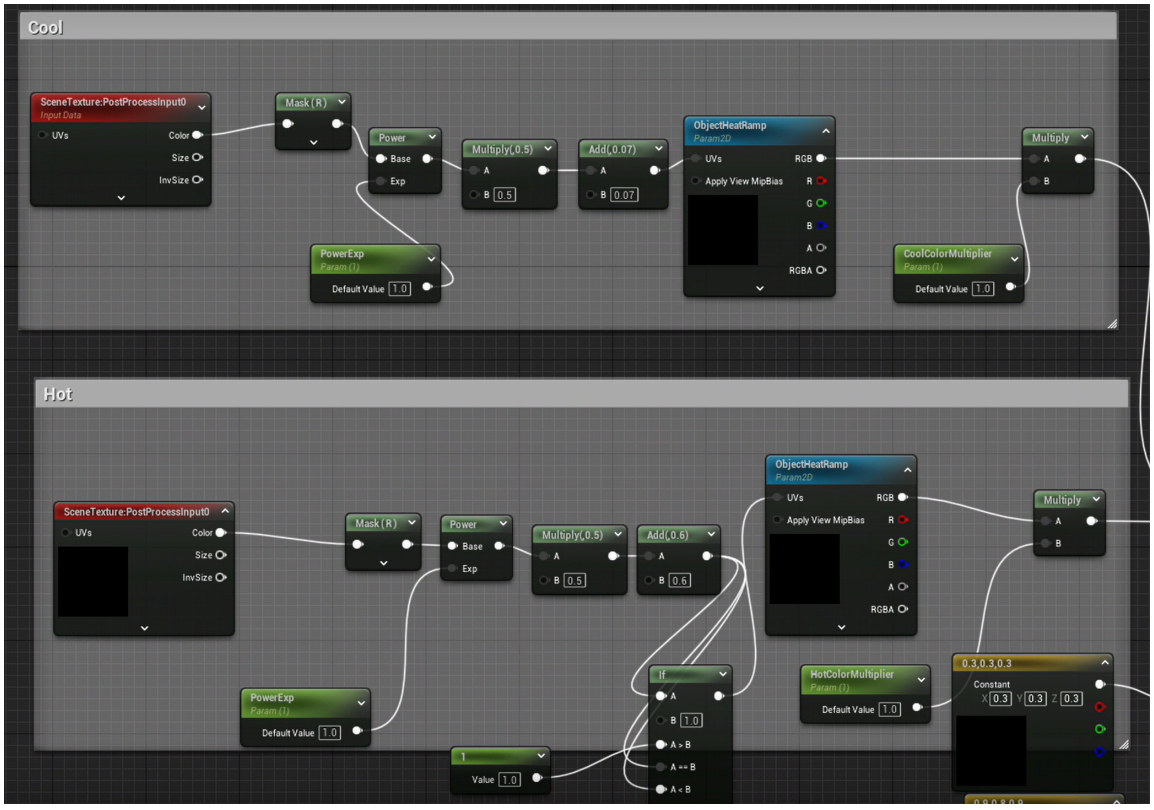


Figure 6.10: 1st thermal material part 1

[2] https://github.com/castacks/firevision_sim

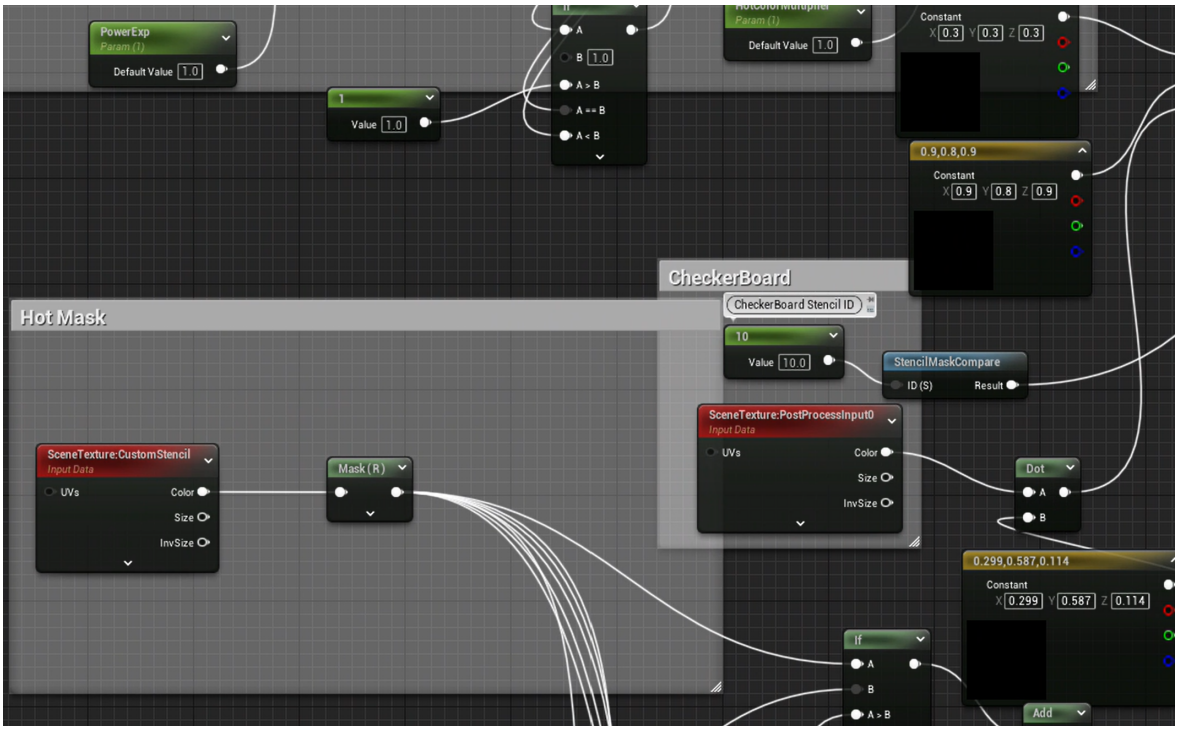


Figure 6.11: 1st thermal material part 2

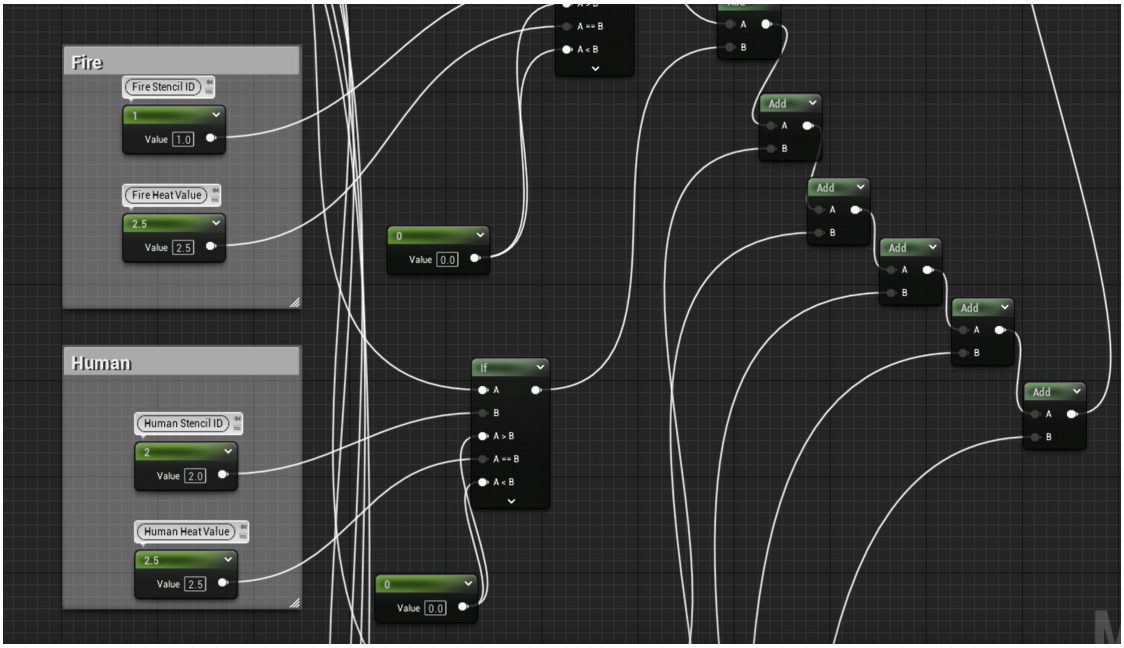


Figure 6.12: 1st thermal material part 3

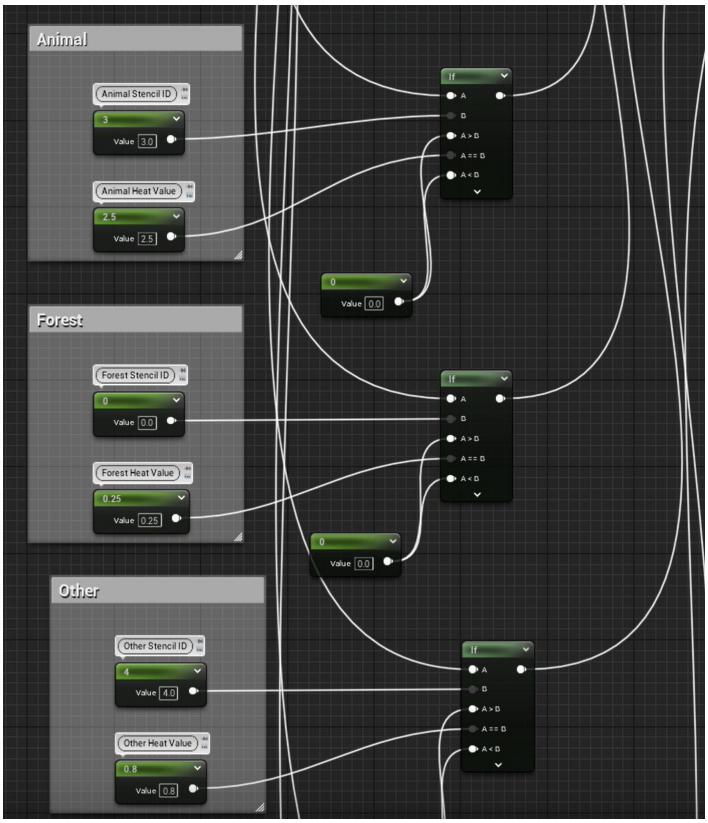


Figure 6.13: 1st thermal material part 4

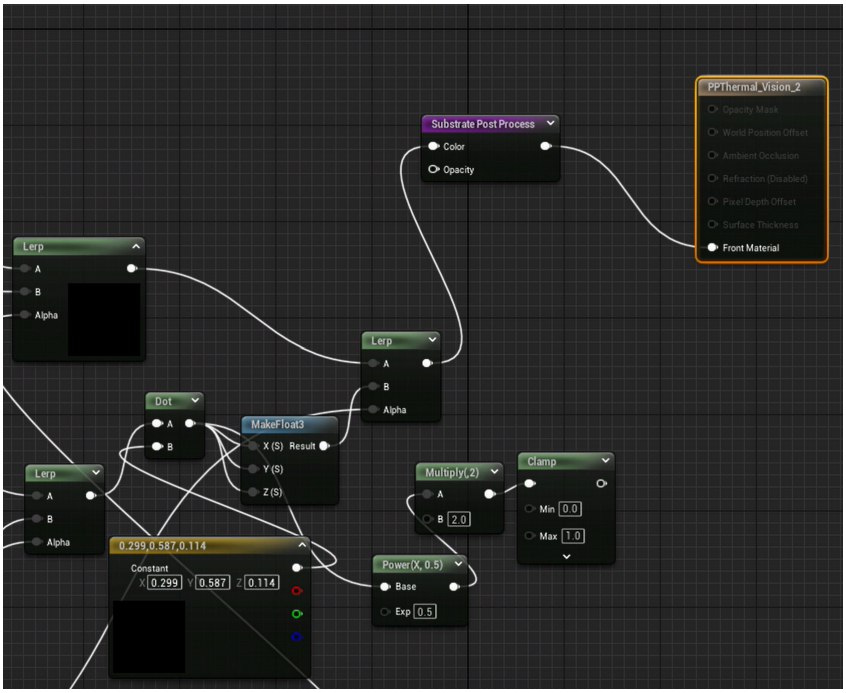


Figure 6.14: 1st thermal material part 5

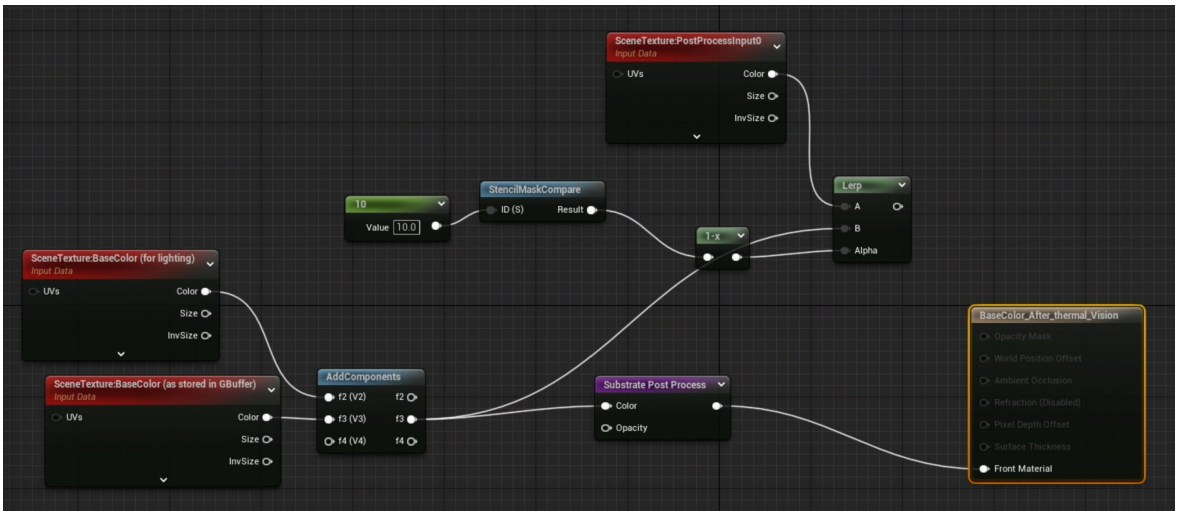


Figure 6.15: 2nd thermal material

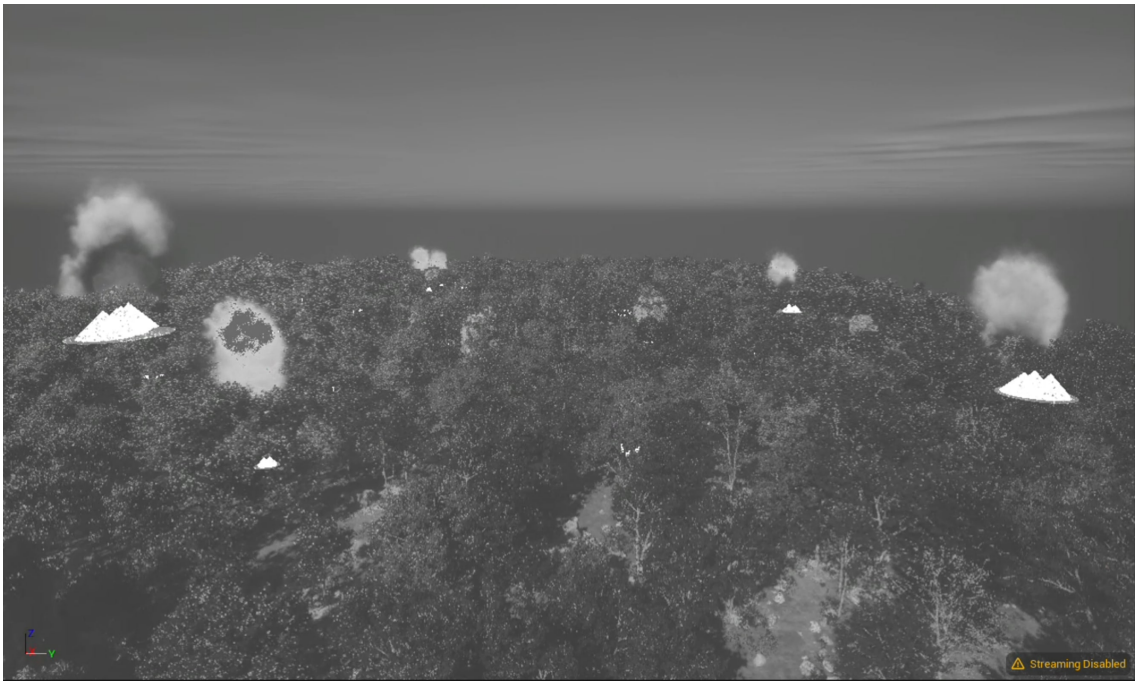


Figure 6.16: Thermal appearance with the first material applied

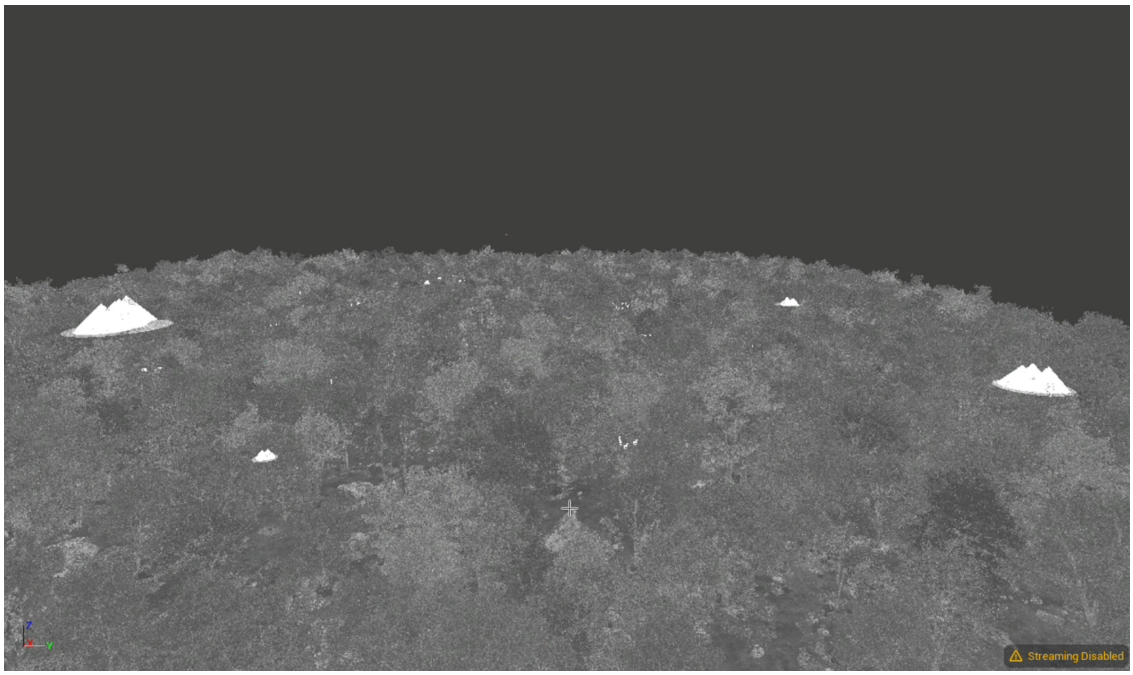


Figure 6.17: Thermal appearance with both materials applied

In order for the material effects to be applied to the objects in the environment, it is necessary to enable the *Custom Depth* rendering option and assign a unique *CustomDepth Stencil Value* to each object. The object categories used for thermal simulation, as shown in Figures 6.11 and 6.12, are defined by their *CustomDepth Stencil Value* assignments as follows: `0 (ID)` for Forest, `1` for Fire, `2` for Human, `3` for Animal, and `4` for Other. These stencil values are used by the post-processing material to assign appropriate thermal colors (temperatures) to each object category. An example of this configuration is shown in Figure 6.18.

Figure 6.18: Custom depth in each object

To apply this sequence of materials, they must be assigned to the camera used by AirSim. This is done by modifying the `BP_PIP_Camera` blueprint located in the Content Drawer under `Plugins` → `Cosys-AirSim Content` → `Blueprints` → `BP_PIP_Camera`, as shown in Figure 6.19. In this blueprint, a new camera must be created to serve as the infrared camera as shown in Figure 6.20. The two post-processing materials are applied to its post-processing volume in the desired order. Additionally, the camera must be configured

with the specific settings required for the simulation, as demonstrated in Figures 6.21, 6.22 and 6.23. Finally, from the settings, we can observe that anti-aliasing is disabled in order to reduce the smoothing effect that can cause pixelized or blurred thermal images. This ensures that the thermal output remains sharp and closer to the desired visual representation.

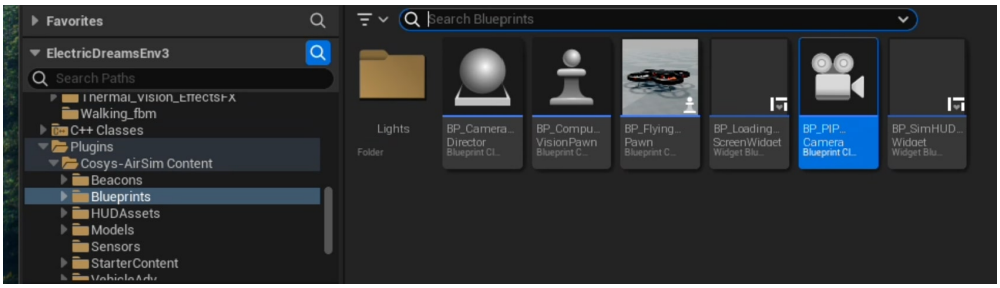


Figure 6.19: BP_PIP_Camera blueprint

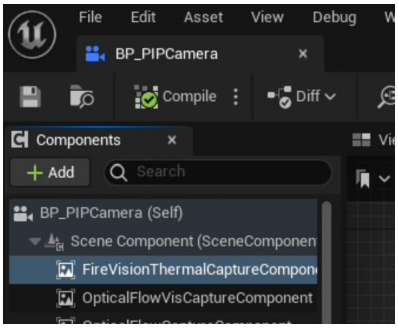


Figure 6.20: Creation of a new AirSim camera

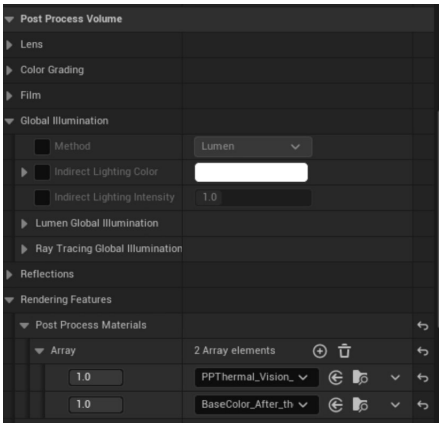


Figure 6.21: Post processing settings

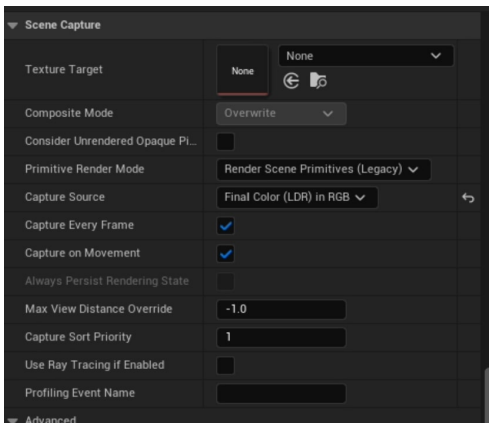


Figure 6.22: Scene capture settings part 1

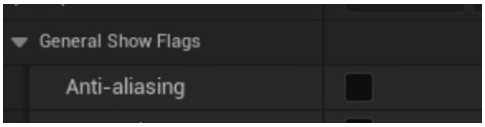


Figure 6.23: Scene capture settings part 2

The thermal appearance of the scene content is shown in the Figures 6.24, 6.25, 6.26 and 6.27 below. As observed, the forest exhibits slight variations in temperature represented by grayscale tones, while other objects such as humans, animals, and fire appear almost entirely white. This design choice was intentional in the first implementation phase, aiming to define a general thermal appearance and evaluate the functionality of the detection system, rather than fully replicate the complex behavior of a real infrared camera, which is considerably more challenging. In addition, the fire is represented by custom cone-shaped meshes. In the RGB camera view, fire is rendered using particle effects, which cannot be directly processed by the thermal camera. Therefore, static cone meshes are used to simulate fire regions. Although this representation does not reflect the physical behavior of the fire, it serves the simulation objectives effectively by providing a consistent thermal target for detection.

Figure 6.24: Thermal appearance of deer

Figure 6.25: Thermal appearance of wolf

Figure 6.26: Thermal appearance of human

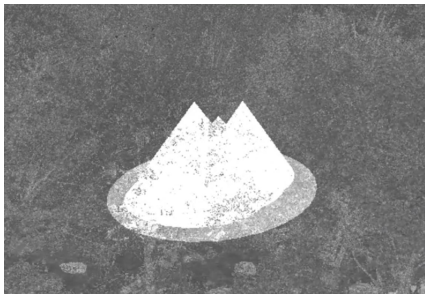


Figure 6.27: Thermal appearance of fire

6.3.3 Thermal Camera Integration with Airsim

For the integration of the custom thermal camera into AirSim, modifications must be made to the source code to allow its configuration through the `settings.json` file. Specifically, a new image type named `FireVisionThermal` was added to enable the use of a custom infrared camera. This image type must be defined and implemented similarly to the standard RGB camera (`Scene`).

The main changes are applied to the `PIPCamera.cpp` file, where the new image type

is registered using the following line:

```
image_type_to_pixel_format_map_.Add(Utils::toNumeric(ImageType::
    FireVisionThermal), EPixelFormat::PF_B8G8R8A8);
```

In addition, the capture component for the new image type is assigned as:

```
captures_[Utils::toNumeric(ImageType::FireVisionThermal)] =
    UAirBlueprintLib::GetActorComponent<USceneCaptureComponent2D>(this,
        TEXT("FireVisionThermalCaptureComponent"));
```

Furthermore, in the image type switch-case block, a new case must be added:

```
case ImageType::FireVisionThermal:
    updateCaptureComponentSetting(captures_[image_type], render_targets_[
        image_type], false,
                                  pixel_format, capture_setting,
                                     ned_transform, false);
    break;
```

These additions must be inserted in the appropriate locations, next to the existing cases for other image types.

Moreover, the image type enumeration in `ImageCaptureBase.hpp` must be extended to include the new image type:

```
enum class ImageType : int {
    Scene = 0,
    DepthPlanar,
    DepthPerspective,
    DepthVis,
    DisparityNormalized,
    Segmentation,
    SurfaceNormals,
    Infrared,
    OpticalFlow,
    OpticalFlowVis,
    FireVisionThermal,
    Annotation,
    Count // must be last
};
```

Although AirSim already includes an `Infrared` image type, it inherits the same configuration as the `Segmentation` camera. As a result, it assigns uniform grayscale values to different objects, which is insufficient for complex environments such as forests, where realistic thermal variation is required. The `FireVisionThermal` type overcomes this limitation by supporting realistic thermal gradients through custom post-processing.

It is essential that the name `FireVisionThermal` used in the C++ code matches the name of the camera created in Unreal Engine as shown in Subsection 6.3.2. This consistency ensures proper linkage between the AirSim backend and the Unreal camera component. In the `settings.json` file, the new thermal camera can now be referenced using `"ImageType": 10`, while the standard RGB camera continues to use `"ImageType": 0`.

Finally, every time changes are made to the source code of the AirSim plugin, it must be rebuilt manually. Once rebuilt, the updated plugin should be placed in the Unreal Engine project directory. Upon opening the project, Unreal Engine will prompt to rebuild the plugin if necessary, allowing seamless integration of the new camera functionality.

In Figure 6.28, we see the final AirSim drone equipped with two cameras, integrated seamlessly with Unreal Engine.

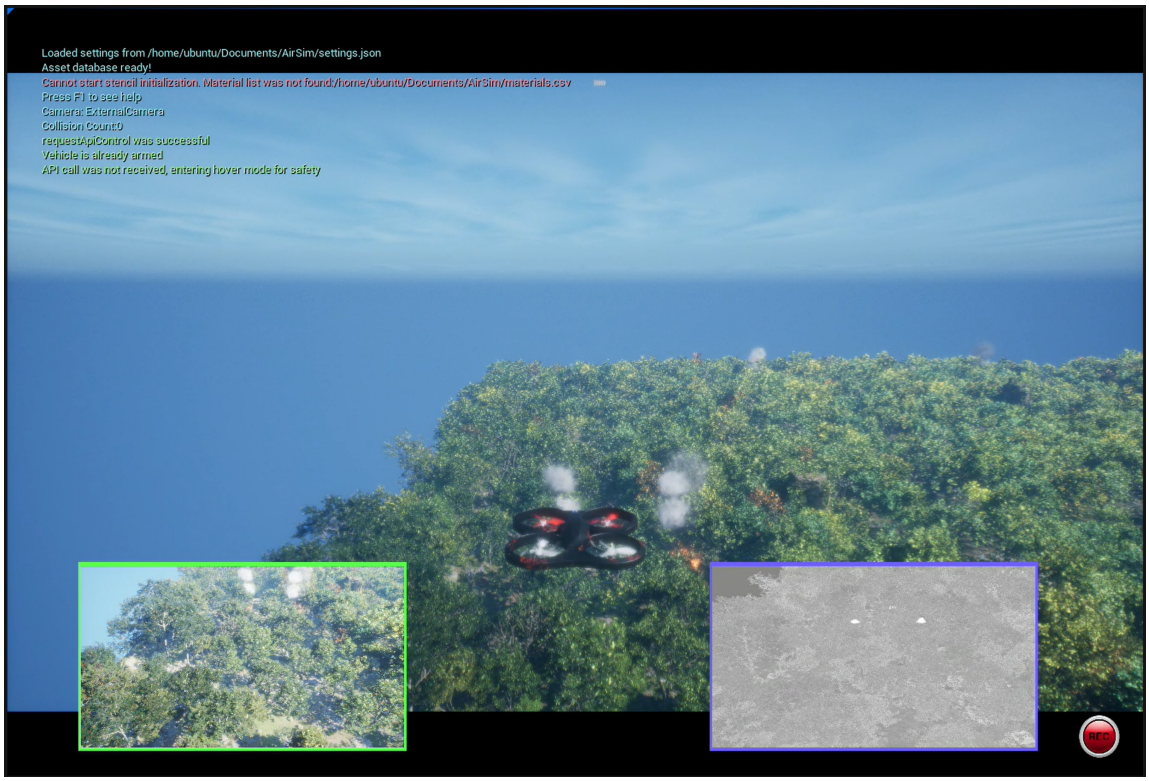


Figure 6.28: AirSim drone integration with cameras and Unreal Engine

Chapter 7

System Evaluation

Introduction

The evaluation of the system was carried out in a simulation environment using *Unreal Engine*, *AirSim* and *ROS 2*. The first step was to initiate the AirSim simulation to setup the required environment. Once AirSim was running, the ROS 2 system was launched to handle the real-time data processing from the drone cameras. As shown in Figure 6.28, two cameras were used for live image streaming. The video feeds from both cameras were processed by the object detection models running on the drone.

The models generated bounding boxes around the detected objects and assigned unique *track IDs* to each object. As seen in the video (YouTube - AirSim and ROS 2 System Simulation), the track IDs remained consistent throughout the operation of the system. However, for *humans* and *animals*, the IDs increased more frequently during the operation of the system. This occurred because the models were less confident in detecting humans and animals, especially in cases involving partial occlusion or low visibility, which led to less accurate predictions.

Data Preparation and Training

Before deploying the system, we collected images for training the models in a dynamic environment, which included various starting positions, and other environmental variations. These images were manually labeled using *Roboflow*, and two datasets were created, each containing *up to 500 images* for each domain. These datasets were crucial for training the models to better detect objects in the simulation environment. This process, known as transfer learning, allowed the models, which had already been trained on real-world data, to adapt and fine-tune their knowledge to the simulation environment, improving their performance and

accuracy. Transfer learning involves taking a model trained on a large dataset with real-world data and applying it to a new but related task or domain, which accelerates the training process and improves the performance of the model on specific tasks.

The thermal domain was observed to pose significant challenges, particularly when detecting humans and animals. Thermal images were highly pixelized, making the analysis more difficult. In addition, custom thermal fire simulations did not accurately represent real-world fire characteristics, making it harder for the model to detect fire in the thermal domain.

To address these challenges, we trained the models in labeled datasets to cover this domain gap (real-world and simulated domains) and then optimized them using *TensorRT with FP16* precision for deployment. This optimization significantly improved the performance of the models, making them more efficient for real-time applications.

System Performance and Results

The results of the optimized models were highly promising, yielding good quality and reliable predictions. As shown in Figures 7.1, 7.2, 7.3, 7.4, and 7.5, two *OpenCV windows* display the real-time feed from the RGB and thermal cameras, with bounding boxes and track IDs corresponding to detected objects. Real-time visualization in the thermal OpenCV window confirms that the image registration and alignment process is successfully applied on-the-fly, ensuring that thermal frames are spatially matched with the corresponding RGB images during system execution.

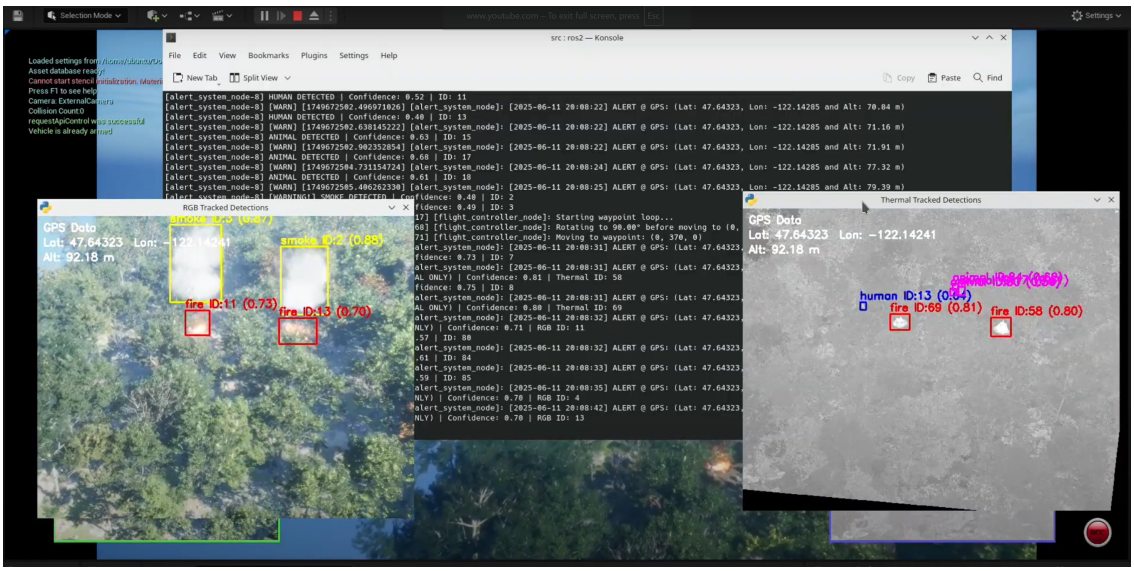


Figure 7.1: Combined RGB and Thermal/IR camera feeds with object detection and tracking (1st example)

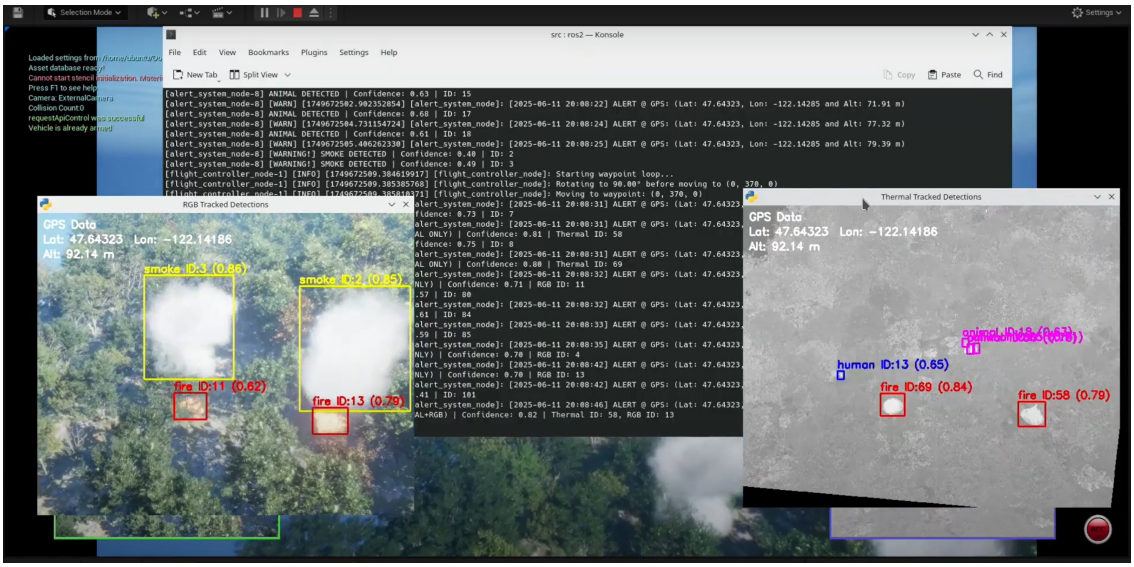


Figure 7.2: Combined RGB and Thermal/IR camera feeds with object detection and tracking (2nd example)

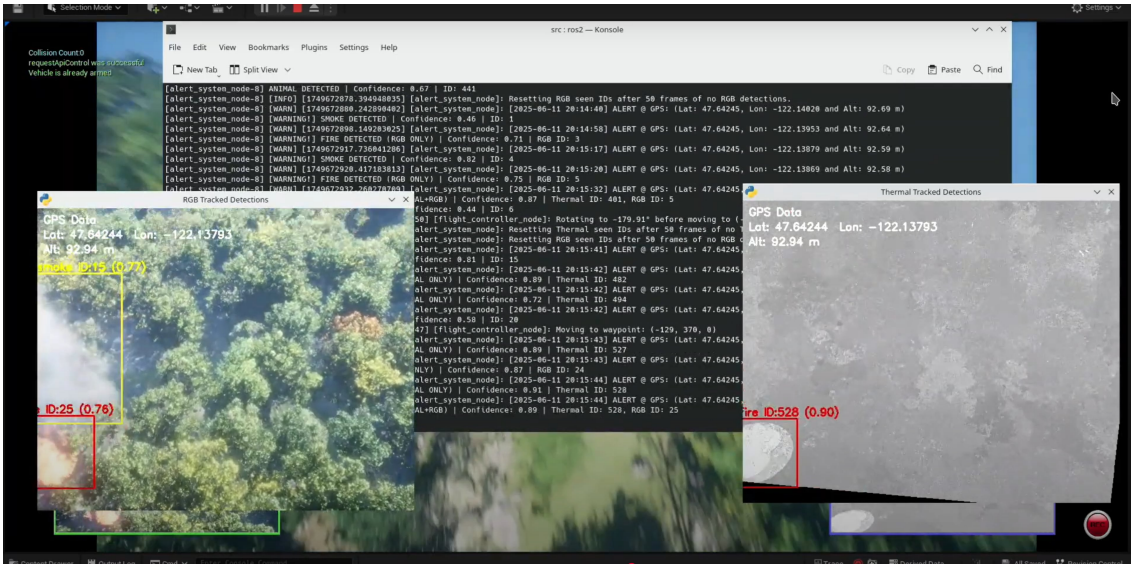


Figure 7.3: Combined RGB and Thermal/IR camera feeds with object detection and tracking (3rd example)

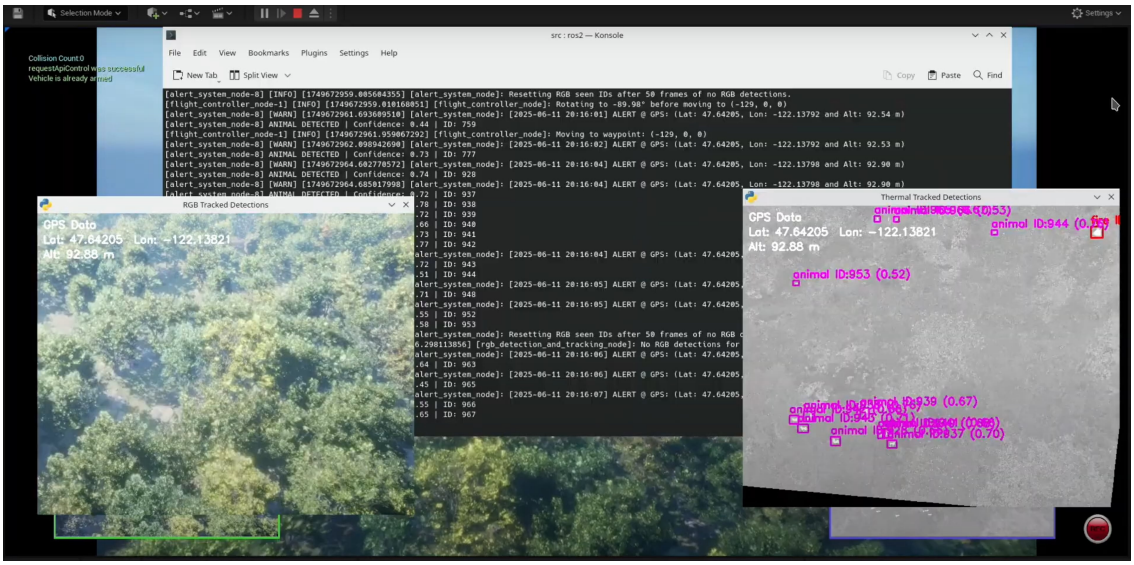


Figure 7.4: Combined RGB and Thermal/IR camera feeds with object detection and tracking (4th example)

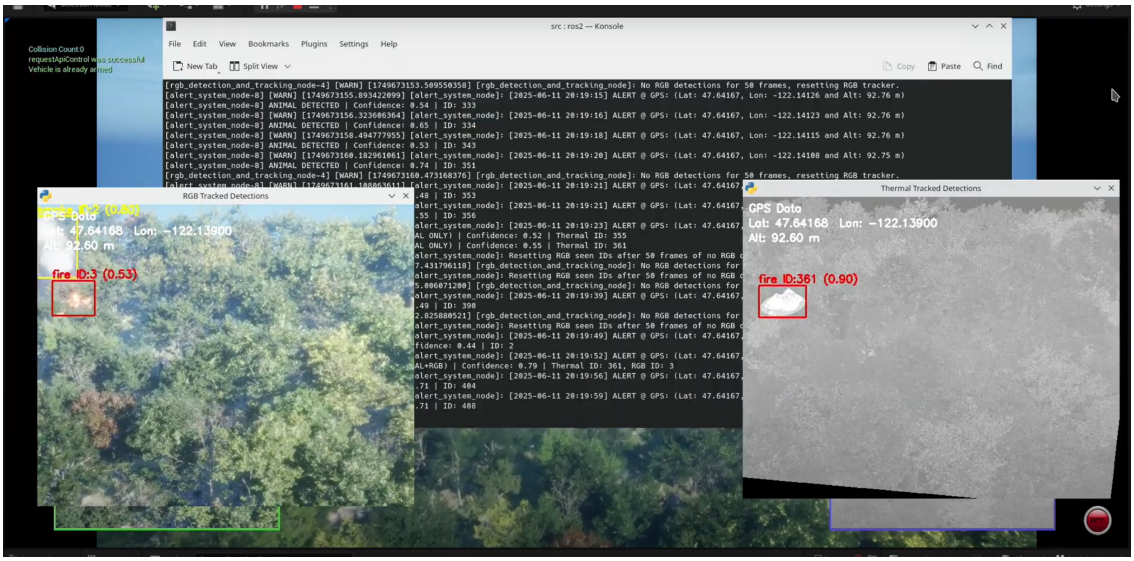


Figure 7.5: Combined RGB and Thermal/IR camera feeds with object detection and tracking (5th example)

Furthermore, the *alerts* for detected *fire*, *smoke*, *human*, and *animal* objects are displayed in the terminal, as seen in the terminal screenshots in Figures 7.6 and 7.7, as well as in previous examples. The detection frequency for humans and animals was significantly higher, as explained earlier, because the models were less confident in these predictions. For fire detection, the system would print the detected type (e.g., fire (*THERMAL ONLY*) or fire (*RGB ONLY*)) based on the modality that made the prediction, with at least a confidence thresh-

old of 0.5 for the thermal domain and 0.7 for the RGB domain. In cases where both RGB and thermal cameras detected the same fire, the system would output both types, such as fire (*THERMAL+RGB*). In Figure 7.7, we can see that the fire in the thermal domain, which is detected in the top right corner immediately triggers the alert of fire in the final record of the terminal (*THERMAL ONLY*). Also, in OpenCV windows we see the GPS coordinates of the drone in real time and in the terminal logs for each prediction. The spatial Intersection over Union (IoU) threshold of 0.5 was used for matching RGB and Thermal fire, which is considered a strict threshold for real-time applications. This strict criterion may have caused some fire alerts from the combination of domains to be missed.

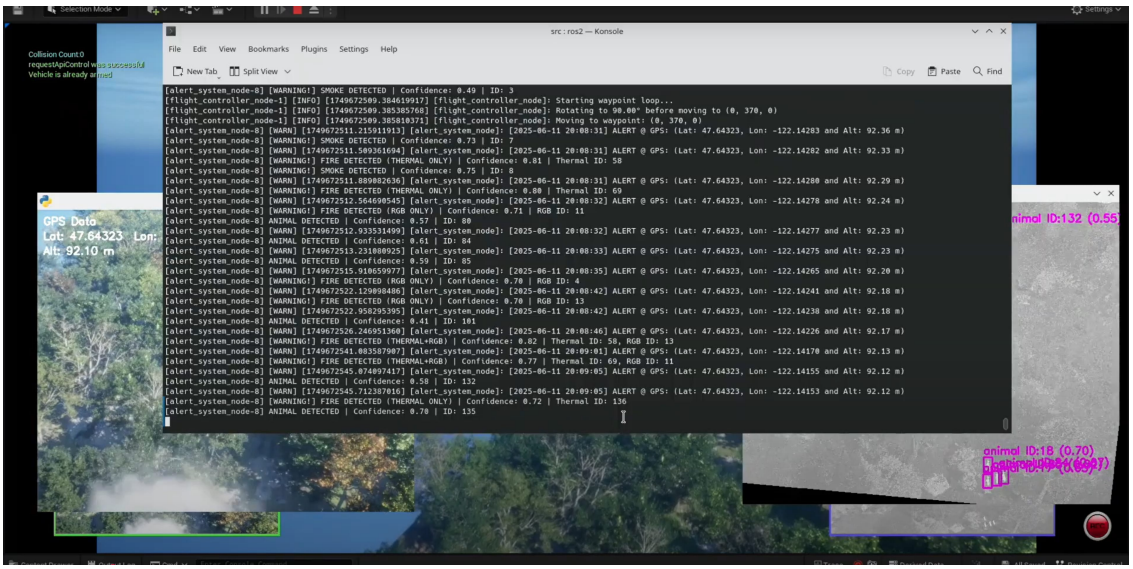


Figure 7.6: Terminal logs showing detection alerts from both domains (1st example)

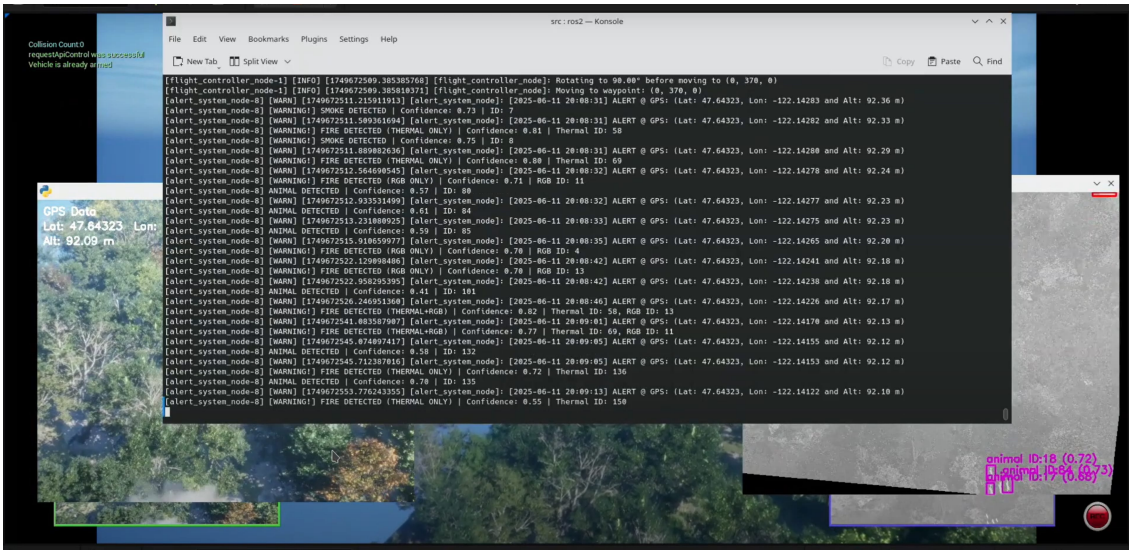


Figure 7.7: Terminal logs showing detection alerts from both domains (2nd example)

The source code for the system, including the models and the training pipeline, has been uploaded to the following GitHub Repository. Please note that datasets and the environment are not included due to their large size (over 100 GB both), which exceeds GitHub's storage limits.

Chapter 8

Conclusions and Future Work

8.1 Summary and Conclusions

This chapter aims to provide a comprehensive summary and reflection of the key results of the system developed throughout the thesis work. This thesis addressed the urgent problem of early wildfire detection by designing and implementing an autonomous UAV-based system that utilizes both RGB and thermal cameras to detect fire, smoke, human, and animal presence in forest environments. The system was built with a modular ROS 2 architecture and it was verified in a realistic simulation environment built in Unreal Engine using AirSim for drone simulation. The innovation of the system is in its continuous/asynchronous multimodal data fusion, and also the combination of the tasks of object detection, tracking, GPS localization, and alerting in a unified software-hardware pipeline.

The datasets used for model training were constructed specifically for this project. The RGB dataset contains annotating fire and smoke pictures that are taken from both synthetic and real-world sources. A lot of data augmentation is done so that the deep learning model does not overfit on the training dataset. On the other hand, the thermal dataset contains annotated fire, human, and animal images that are the only source of footage from the real world. A key goal of this dataset (having also humans and animals annotations) was to help the model distinguish humans from animals using thermal profiles.

For the model evaluation, the most suitable one was found among YOLOv8s, EfficientDet D1, SSD with VGG16, and Faster R-CNN with MobileNetV3, by conducting a variety of experiments over RGB as well as thermal modalities. The procedure of assessment included the following points: training/validation loss, validation metrics, inference performance and

resource consumption. In the end, YOLOv8s was the model that performed best in both domains and was chosen. Concerning the RGB dataset, YOLOv8s is the most accurate model for pursuit of the fire and smoke classes, while having the least time for inference and memory utilization. For thermal dataset, YOLOv8s again appeared as the winner among the other models in the tasks of fire, human, and animal detection, giving a high classification rate for true positives, and generating the least amount of false alarms while being the fastest and most robust model.

To meet real-time constraints on embedded hardware, the selected YOLOv8s model was optimized using NVIDIA TensorRT, and tested with FP16 and INT8 precision modes. The INT8 model version achieved a great speedup in inference time without sacrificing detection accuracy almost identical. These performance improvements make it possible to install on edge devices such as the Jetson Orin Nano and have high-performance detection that is still feasible in such constrained onboard environments.

A significant contribution of this thesis was the implementation of a complete ROS 2-based detection pipeline. Every single function was written in the ROS 2-dedicated node, for example, a sensor data collector node assures that the data are synchronized between RGB, thermal, and GPS. Besides that, a preprocessing node performs the registration of images, whereas detection nodes execute the optimized YOLOv8s models. A method of homography-based image registration to align RGB and thermal frames has been presented in this thesis. A fusion node has been used to combine the two modalities' outputs, while the tracking technique in detection nodes such as DeepSORT, BoT-SORT and ByteTrack helps to keep the object identity over time enabling thus the system to send out coherent, non-redundant alerts. An alert node is in charge of the filtered detection events and it goes even further by adding GPS coordinates for those events to be used downstream. Finally, a visualization node displayed annotated image streams for offline analysis.

The system was evaluated in a rich simulation environment developed in Unreal Engine. The environment included forests, animals, fire, smoke, and human characters, allowing for thorough testing in scenarios that approximate real-world complexities. A thermal camera was custom-built using post-processing materials and was registered as a sensor in AirSim.

The findings in Chapter 7 indicated that the whole system is functional in real-time, continuously giving 10 FPS with no loss in performance. It managed to identify and keep track of the objects correctly, even when they were partly hidden or visibility was bad. The alert

generation was consistent, and each of the detections was accompanied by accurate geolocation information. Visual alignment between thermal and RGB frames confirmed the success of the homography-based registration and validated the multimodal fusion approach.

The thesis also demonstrates the successful deployment of a real-time wildfire detection pipeline on embedded hardware. The system was validated on the NVIDIA Jetson Orin Nano platform using optimized YOLOv8s models. This confirms the feasibility of deploying the system on a UAV for fully autonomous missions in remote environments, where low-latency processing and energy efficiency are crucial. The high frame rate and stable performance make it a strong candidate for integration in practical wildfire monitoring systems that must operate in constrained on-board computing environments.

Another key conclusion is the validation of a reversed, but effective, simulation-to-real pipeline. Instead of the more common approach of training on synthetic data and adapting to real-world deployment, this thesis adopts the opposite route: training on real-world data and validating through simulation with a small transfer learning. Despite the complexity of this strategy, the rich simulation environment created in Unreal Engine, with dynamic forests, fire, smoke, human, and animal assets, enabled controlled and repeatable experiments that reflected real-world conditions. The successful alignment of RGB and thermal views through homography and the accurate fusion of detections demonstrated that conclusions about system performance in real time could indeed be drawn from simulation-based testing, thereby establishing a strong foundation for future field deployment.

Furthermore, the inclusion of thermal cameras significantly enhances detection capabilities, allowing the system to identify critical classes such as humans and animals, which are impossible to detect reliably using RGB imagery alone at UAV operating altitudes.

In addition, the fusion of thermal and RGB detections, enabled by accurate image registration, increased the overall detection robustness. This multimodal approach ensured reliable fire and object detection even under challenging environmental conditions such as smoke, partial occlusions, and varying lighting.

Finally, this thesis successfully developed, implemented, and validated a UAV-based wildfire detection system that combines RGB and thermal imagery through deep learning models. It demonstrated how modular design, real-time optimizations, and simulation-driven testing can lead to a robust and deployable platform for environmental monitoring. The choice of YOLOv8s for the detection, the efficient incorporation of TensorRT optimization, and the

construction of an end-to-end ROS 2 system highlight the realistic practical viability of the approach and these findings open the door to further research.

8.2 Future Work

A number of exciting research prospects and system improvement opportunities are noted in conclusion to this thesis contribution and its implementation work. The proposed system extensions are targeted at not only extending the performance and reliability of the UAV-based wildfire detection system but also supporting its transition from an experimental stage to practical applications while still significantly increasing its capabilities.

Firstly, a new project is proposed that aims the current YOLOv8s architecture to be modified to include an internal multimodal fusion mechanism. Instead of processing the RGB and thermal images separately with two independent detection and tracking pipelines, the proposed approach would integrate early or late fusion layers directly within a single detection and one tracking model. This upgrade would not only reduce the computational overhead and get rid of the parts that are repeated but also simplify the overall system structure by removing one of the detection and tracking models. The new system based on fusion-aware detection architecture will be a matter of further work and detailed examination, as referred to in Chapter 3.

Another critical enhancement involves the incorporation of LiDAR sensor into the UAV system. While the current system do not calculate fire and human coordinates, a LiDAR integration would provide more precise depth information. This could enable accurate 3D localization of detected objects (fire, smoke, humans) and significantly improve the geospatial accuracy of the generated alerts.

Similarly, the UAV should adopt a fire-centric observation behavior when a fire is detected. Instead of immediately continuing its path, the UAV should adjust its trajectory to center the fire within the camera frame and remain stationary or hover for approximately 10 seconds. This would allow the capturing of several consecutive image frames from various perspectives, thus enhancing the detection confidence and allowing a more accurate evaluation of the fire spread.

Furthermore, human detections in remote areas may indicate either victims or potentially suspicious activity. The system could also be made to track and follow the persons detected

within a limited area of patrol to provide persistent surveillance and security applications. This patrolling behavior would ensure that individuals are monitored within a predefined perimeter, assisting in early identification of criminal actions such as arson.

Lastly, next steps should be directed towards moving from the present simulation-based platform to a fully functioning UAV system with real sensors and onboard computation. The whole architecture, comprising of various components such as onboard inference, sensor synchronization, image registration, alert generation and others may indeed be launched and tried out in the open field. This kind of end-to-end deployment will facilitate a complete assessment of the system's stability, response time, cooling system, and flight control in outdoor conditions that are not controlled, hence extending the validation of the proposed wildfire monitoring framework to the practical aspect.

Bibliography

[1] What is machine learning? https://www.datacamp.com/blog/what-is-machine-learning.

[2] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016.

[3] What is deep learning? https://www.ibm.com/think/topics/deep-learning.

[4] What is Object Detection? A Comprehensive Guide. https://zilliz.com/learn/what-is-object-detection.

[5] Ayoub Benali Amjoud and Mustapha Amrouch. Object Detection Using Deep Learning, CNNs and Vision Transformers: A Review. *IEEE Access*, PP:1–1, 01 2023.

[6] Non Maximum Suppression: Theory and Implementation in PyTorch. https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/.

[7] What is Object Tracking in Computer Vision? https://blog.roboflow.com/what-is-object-tracking-computer-vision/.

[8] A complete overview of Object Tracking Algorithms in Computer Vision & Self-Driving Cars. https://www.thinkautonomous.ai/blog/object-tracking/.

[9] Hamilton Adoni, Sandra Lorenz, Junaidh Fareedh, Richard Gloaguen, and Michael Bussmann. Investigation of Autonomous Multi-UAV Systems for Target Detection in Distributed Environment: Current Developments and Open Challenges. *Drones*, 7:263, 04 2023.

[10] RGB cameras: Definition, components, and integration. https://www.technexion.com/resources/rgb-cameras/.

[11] What is Infrared? https://www.flir.com/discover/what-is-infrared/.

[12] Martin Brenner, Napoleon Reyes, Teo Susnjak, and Andre Barczak. RGB-D and Thermal Sensor Fusion: A Systematic Literature Review. *IEEE Access*, PP:1–1, 01 2023.

[13] The Basics of Spectral Bands: NIR, SWIR, and RGB. https://swiftgeospatial.solutions/2025/03/12/the-basics-of-spectral-bands-nir-swir-and-rgb/.

[14] The Differences Between SWIR, MWIR, and LWIR Cameras. https://www.axiomoptics.com/blog/the-differences-between-swir-mwir-and-lwir-cameras/.

[15] Far Infrared (FIR). https://acktar.com/far-infrared-fir/.

[16] What is the Thermal Infrared Range? And What are NIR, SWIR, MWIR and LWIR? https://www.ametek-land.com/pressreleases/blog/2021/june/thermalinfraredrangeblog.

[17] What is GPS and how does it work? https://novatel.com/support/knowledge-and-learning/what-is-gps-gnss.

[18] Unreal Engine 5 editor. https://www.unrealengine.com/en-US/blog/unreal-engine-5-1-is-now-available.

[19] Cosys-AirSim. https://github.com/Cosys-Lab/Cosys-AirSim/tree/5.2.1.

[20] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. *CoRR*, abs/1705.05065, 2017.

[21] ROS 2 (Robot Operating System): overview and key points for robotics software. https://robotnik.eu/ros-2-robot-operating-system-overview-and-key-points-for-robotics-software/.

[22] Xiwen Chen, Bryce Hopkins, Hao Wang, Leo O'Neill, Fatemeh Afghah, Abolfazl Razi, Peter Fulé, Janice Coen, Eric Rowell, and Adam Watts. Wildland Fire Detection and Monitoring Using a Drone-Collected RGB/IR Image Dataset. *IEEE Access*, 10:121301–121317, 2022.

[23] Bryce Hopkins, Leo O'Neill, Fatemeh Afghah, Abolfazl Razi, Eric Rowell, Adam Watts, Peter Fule, and Janice Coen. FLAME 2: Fire detection and modeLing: Aerial Multi-spectral imagE dataset, 2022.

[24] S.D.M.W. Kularatne, Constantino Álvarez Casado, Janne Rajala, Tuomo Hänninen, Miguel Bordallo López, and Le Nguyen. FireMan-UAV-RGBT: A Novel UAV-Based RGB-Thermal Video Dataset for the Detection of Wildfires in the Finnish Forests. In *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2024.

[25] Abdelmalek Bouguettaya, Hafed Zarzour, Amine Mohammed Taberkit, and Ahmed Kechida. A review on early wildfire detection from unmanned aerial vehicles using deep learning-based computer vision algorithms. *Signal Processing*, 190:108309, 2022.

[26] Mukhriddin Mukhiddinov, Akmalbek Bobomirzaevich Abdusalomov, and Jinsoo Cho. A Wildfire Smoke Detection System Using Unmanned Aerial Vehicle Images Based on the Optimized YOLOv5. *Sensors*, 22(23), 2022.

[27] Donghua Wu, Zhongmin Qian, Dongyang Wu, and Junling Wang. FSNet: Enhancing Forest-Fire and Smoke Detection with an Advanced UAV-Based Network. *Forests*, 15(5), 2024.

[28] Armando M Fernandes, Andrei B Utkin, and Paulo Chaves. Automatic early detection of wildfire smoke with visible-light cameras and EfficientDet. *Journal of Fire Sciences*, 41(4):122–135, 2023.

[29] Exploring Data Labeling and the 6 Different Types of Image Annotation. https://xailient.com/blog/exploring-data-labeling-and-the-6-different-types-of-image-annotation/.

[30] Object Detection: COCO JSON formats. https://blog.stackademic.com/object-detection-coco-json-formats-a27e52a8b0ee.

[31] gaiasd. DFireDataset. https://github.com/gaiasd/DFireDataset, 2022.

[32] AUIC. fire Dataset . https://universe.roboflow.com/auic-3tgy0/fire-hsy0i, 2022.

[33] Yunnan University. synthetic fire-smoke Dataset . https://universe.roboflow.com/yunnan-university/synthetic-fire-smoke, 2023.

[34] Windula Kularatne, Kamand Sedaghat Shayegan, Constantino Álvarez Casado, Janne Rajala, Tuomo Hänninen, Miguel Bordallo López, and Ngu Le Nguyen. FireMan-UAV-RGBT, September 2024.

[35] Anam Ibn Jafar, Al Mohimanul Islam, Fatiha Binta Masud, Jeath Rahmat Ullah, and Md. Rayhan Ahmed. FlameVision: A new dataset for wildfire classification and detection using aerial imagery, 2023.

[36] Thermal Disasters Project. Thermal Human Detection from UAV Dataset . https://universe.roboflow.com/thermal-disasters-project/thermal-human-detection-from-uav, 2023.

[37] Oscar Ramirez. Data set of thermal images of people in forested areas. 11 2023.

[38] Project FieldEye. Project FieldEye Dataset . https://universe.roboflow.com/project-fieldeye/project-fieldeye, 2022.

[39] Tan Tram. SAR-TRAIN-O Dataset . https://universe.roboflow.com/tan-tram/sar_train_o, 2021.

[40] Elizabeth Bondi, Raghav Jain, Palash Aggrawal, Saket Anand, Robert Hannaford, Ashish Kapoor, Jim Piavis, Shital Shah, Lucas Joppa, Bistra Dilkina, and Milind Tambe. BIRDSAI: A Dataset for Detection and Tracking in Aerial Thermal Infrared Videos. In *WACV*, 2020.

[41] bootle. Thermal Fire Dataset . https://universe.roboflow.com/bootle/fire-khebn, 2023.

[42] RGB-Thermal Wildfire dataset . https://complex.ustc.edu.cn/sjwwataset/list.htm.

[43] firestereo. FIReStereo. https://github.com/firestereo/firestereo, 2025.

[44] object detection. infrared-fire Dataset . https://universe.roboflow.com/object-detection-istuk/infrared_fire, 2024.

[45] Alireza Shamsoshoara, Fatemeh Afghah, Abolfazl Razi, Liming Zheng, Peter Fulé, and Erik Blasch. The FLAME dataset: Aerial Imagery Pile burn detection using drones (UAVs), 2020.

[46] Bryce Hopkins, Leo O'Neill, Fatemeh Afghah, Abolfazl Razi, Eric Rowell, Adam Watts, Peter Fule, and Janice Coen. FLAME 2: Fire detection and modeLing: Aerial Multi-spectral imagE dataset, 2022.

[47] Uni. Wolf Dataset . https://universe.roboflow.com/uni-p9rzq/wolf-0epk1, 2024.

[48] Ed. Thermal Deer Dataset . https://universe.roboflow.com/ed-e8koj/thermal-deer-sgyfo, 2023.

[49] ItamarM. Thermal Animals Dataset . https://universe.roboflow.com/itamarm/new-data-tvtkb, 2024.

[50] hongwai. Thermal Animals Dataset . https://universe.roboflow.com/hongwai/animal-4kv9s, 2024.

[51] object detection. Thermal Animals Dataset . https://universe.roboflow.com/object-detection-d3ovr/thermal-animals-thdnz, 2024.

[52] Project1. Thermal Deer Dataset . https://universe.roboflow.com/project1-pjv7c/deer-model-gqiqk, 2025.

[53] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*, abs/1506.02640, 2015.

[54] What is YOLOv8? A Complete Guide. https://blog.roboflow.com/what-is-yolov8/.

[55] Under the Hood: YOLOv8 Architecture Explained. https://keylabs.ai/blog/under-the-hood-yolov8-architecture-explained/.

[56] Wenning Zhao, Xin Yao, Bixin Wang, Jiayi Ding, Jialu Li, Xiong Zhang, Shuting Wan, Jingyi Zhao, Rui Guo, and Wei Cai. A Visual Detection Method for Train Couplers Based on YOLOv8 Model. In Saman K. Halgamuge, Hao Zhang, Dingxuan Zhao, and Yongming Bian, editors, *The 8th International Conference on Advances in Construction Machinery and Vehicle Engineering*, pages 561–573, Singapore, 2024. Springer Nature Singapore.

[57] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3. *CoRR*, abs/1905.02244, 2019.

[58] Siying Qian, Chenran Ning, and Yuepeng Hu. MobileNetV3 for Image Classification. In *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 490–497, 2021.

[59] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*, abs/1506.01497, 2015.

[60] The Fundamental Guide to Faster R-CNN. https://viso.ai/deep-learning/faster-r-cnn-2/.

[61] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[62] Very Deep Convolutional Networks (VGG) Essential Guide. https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/.

[63] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. *CoRR*, abs/1512.02325, 2015.

[64] Mingxing Tan, Ruoming Pang, and Quoc V. Le. EfficientDet: Scalable and Efficient Object Detection. *CoRR*, abs/1911.09070, 2019.

[65] SGD. https://docs.pytorch.org/docs/stable/generated/torch.optim.SGD.html.

[66] Cosine Learning Rate Schedulers in PyTorch. https://medium.com/@utkrisht14/cosine-learning-rate-schedulers-in-pytorch-486d8717d541.

[67] What is Early Stopping in Deep Learning? https://cyborgcodes.medium.com/what-is-early-stopping-in-deep-learning-eeb1e710a3cf.

[68] Understanding Nvidia TensorRT for deep learning model optimization. https://medium.com/@abhaychaturvedi_72055/understanding-nvidias-tensorrt-for-deep-learning-model-optimization-dad3eb6b26d9.

[69] Defining Floating Point Precision - What is FP64, FP32, FP16? https://www.exxactcorp.com/blog/hpc/what-is-fp64-fp32-fp16.

[70] Camera Modeling: Exploring Distortion and Distortion Models, Part I. https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i.

[71] An Introduction to BYTETrack: Multi-Object Tracking by Associating Every Detection Box. https://www.datature.io/blog/introduction-to-bytetrack-multi-object-tracking-by-associating-every-detection-box.

[72] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. ByteTrack: Multi-Object Tracking by Associating Every Detection Box. *CoRR*, abs/2110.06864, 2021.

[73] Mastering Deep Sort: The Future of Object Tracking Explained. https://www.ikomia.ai/blog/deep-sort-object-tracking-guide.

[74] Addie Ira Borja Parico and Tofael Ahamed. Real Time Pear Fruit Detection and Counting Using YOLOv4 Models and Deep SORT. *Sensors*, 21(14), 2021.

[75] Yunxiang Liu and Shujun Shen. Vehicle Detection and Tracking Based on Improved YOLOv8. *IEEE Access*, PP:1–1, 01 2025.

[76] Lu Wang, K. Law, Chan Tong Lam, Benjamin Ng, Wei Ke, and Marcus Im. Automatic Lane Discovery and Traffic Congestion Detection in a Real-Time Multi-Vehicle Tracking Systems. *IEEE Access*, PP:1–1, 01 2024.