Ministry of Education and Science of Ukraine

National University of "Kyiv-Mohyla Academy"

Department of Mathematics of the Faculty of Informatics

Optimization Methods for Hyperparameter Selection in UAV Object Detection Systems

Text part to the thesis in the specialty "Applied Mathematics" - 113

Thesis Supervisor:

Senior Lecturer

Andrew Kurochkin

_____ (Signature)

"___" _____ 2026

Done by Student:

Mariia Cherkasova

"___" _____ 2026

Kyiv, 2026

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"
Department of Mathematics of the Faculty of Informatics

Approved
Head of Department of Mathematics,
Associate Professor, Ph.D.
R. K. Chornei

_____ (Signature)

"___" _____ 2026

INDIVIDUAL TASK for thesis
of the student Mariia Cherkasova
4th year of the Faculty of Informatics
topic: Optimization Methods for Hyperparameter Selection in UAV Object Detection Systems

The content of the PM to the thesis:
Abstract
Introduction
Related work and Algorithms Overview
Detection Model Selection
Approach
Evaluation
Further work and Conclusions
Bibliography

Date of issue: "___" _____ 2026

Supervisor: _____ (Signature)

Task received: _____ (Signature)

Schedule for preparation for the defence of the qualification work

Schedule approved «_____» _____ 2025y.

№ з/п	List of tasks	The deadline for the phase	Signature of the supervisor	Date of supervisor's review	Comment
1.	Obtaining a topic for a qualification work.	19.09.2025			
2.	Familiarization with the topic for a qual- ification work.	30.09.2025			
3.	Development of work's plan and struc- ture.	06.10.2025			
4.	Study of relevant literature. Writing an introduction.	20.12.2025			
5.	Performing a series of experiments to pre-train the model.	20.01.2026			
6.	Work on the textual design of the theoretical part and the obtained results.	20.02.2026			
7.	Preliminary analysis of qualification work.	03.03.2026			
8.	Preliminary defence of qualification work.	02.04.2026			
9.	Defence of qualification work.	28.05.2026			

Supervisor: Kurochkin Andrew Volodymyrovych

Student: Cherkasova Mariia Dmytrivna

Abstract

This thesis investigates hyperparameter optimization methods for object detection models deployed on a UAV platform. Detection in aerial imagery is challenged by varying object scales, complex backgrounds, sensor-induced image degradation, and motion complexity — the simultaneous movement of targets, the camera gimbal, and the UAV platform itself. These factors make model performance highly sensitive to hyperparameter configuration.

The work was conducted on an existing UAV object detection codebase using a multi-source dataset comprising aerial and ground-level imagery of military and civilian objects. The contributions include implementing a two-stage hyperparameter optimization pipeline using Optuna — combining random search with a subsequent TPE-based optimization — as well as redesigning the data preprocessing pipeline.

The HPO-tuned model achieved mAP@0.5 = 0.623 compared to 0.564 for the baseline (+10.46%). The F1 score improved from 0.51 to 0.58 (+13.73%). Statistical significance was confirmed using a Wilcoxon signed-rank test ($p = 0.004$, $\alpha = 0.05$, Cohen's $d = 0.693$).

Keywords: computer vision, object detection, hyperparameter optimization, UAV imagery, YOLOv8, Bayesian optimization, mean average precision.

Анотація

Дана дипломна робота досліджує методи оптимізації гіперпараметрів для моделей виявлення об'єктів на платформі БПЛА. Виявлення об'єктів на аерофотознімках ускладнюється мінливими масштабами, складним фоном, деградацією зображення та динамікою руху — одночасним переміщенням об'єктів, камери та платформи. Це робить модель чутливою до конфігурації гіперпараметрів.

Робота виконувалася на основі наявної структури коду виявлення об'єктів з БПЛА на багатоджерельному наборі даних військових та цивільних об'єктів. Реалізовано двоетапну оптимізацію гіперпараметрів за допомогою Optuna: випадковий пошук з подальшою TPE-оптимізацією, та переробку конвеєра обробки даних.

Модель з оптимізованими гіперпараметрами досягла mAP@0.5 = 0.623 проти 0.564 базової (+10.46%). Показник F1 зріс з 0.51 до 0.58 (+13.73%). Значущість підтверджена критерієм Вілкоксона ($p = 0.004$, $\alpha = 0.05$, d Коена = 0.693).

Ключові слова: комп'ютерний зір, виявлення об'єктів, оптимізація гіперпараметрів, аерофотознімки БЛА, байєсівська оптимізація, середня точність.

Table of Contents

Introduction

Background and Motivation

Recent advances in deep learning have made it possible to detect objects in images with high accuracy and at real-time speed. This has opened the door to deploying detection models directly on edge devices, including drones. However, UAV-based detection poses additional challenges compared to standard settings: models must operate under strict latency and hardware constraints while handling aerial-specific difficulties such as small object size, altitude variation, and unstable imaging conditions.

Over the past few years, the need to improve real-time object detection algorithms and adapt them for use on UAVs has grown significantly. This is due to the increasing deployment of such systems in applications that directly affect human lives. Drones with onboard real-time object detection are used in both civilian and military contexts: extinguishing fires, evacuating people from disaster sites, conducting reconnaissance operations, and identifying military targets. These use cases highlight the importance of reliable algorithm performance and the ability to process visual information within milliseconds — making the quality of object detection a critical factor.

Given the ongoing wartime situation in Ukraine, this is especially critical. Autonomous UAVs would reduce the need for human presence on the battlefield, as drones would be able to independently perform assigned tasks without a pilot. Furthermore, automated recognition of military objects can produce more consistent and predictable results in reconnaissance compared to human operators, who may suffer from fatigue, exhaustion, and stress. In general, high-quality real-time detection would bring UAV autonomy to a level where decisions can be made effectively without human intervention.

Objectives and Scope

This work focuses on hyperparameter optimization for a real-time object detection model trained on a custom multi-source dataset of military and civilian objects.

Training a deep learning model involves two levels of optimization. The first is the training process itself, where the model minimizes a loss function to learn from the data. The second is hyperparameter selection — choosing the configuration under which the

model trains best. This work focuses on the second level: finding the hyperparameter values that lead to the highest detection accuracy on a custom dataset.

The first stage of the work covers a review of existing object detection approaches and hyperparameter optimization methods, with a particular focus on methods applicable to detection on aerial imagery. This includes an analysis of existing hyperparameter optimization algorithms and a justification of the approach that was selected for this work. The next stage is practical: the training pipeline is prepared and refined to ensure data quality and consistency before any optimization is performed. This is followed by the implementation of a two-stage hyperparameter search algorithm, which involves multiple training iterations before the final configuration is selected. The best-performing model is then evaluated on a held-out test set using standard detection metrics. Finally, statistical testing is applied to confirm that the improvement over the baseline model is significant and not a result of random variation.

Thesis structure

The introduction chapter provides the background, motivation, objectives and scope of the work. Chapter 1 covers the theoretical background: the evolution of object detection approaches, the specifics of detection on aerial imagery, and a review of hyperparameter optimization algorithms. Chapter 2 is dedicated to model selection, covering the motivation behind the choice, the model architecture, and the loss functions used during training. Chapter 3 describes the practical approach: the training pipeline setup and the hyperparameter optimization process. Chapter 4 presents the results of the evaluation, including metric-based comparison of the baseline and HPO-tuned models and human evaluation findings. The last chapter summarizes the conclusions and outlines directions for further work.

Chapter 1

Related Work and Algorithms Overview

Computer vision, as a big field in artificial intelligence, contains several subfields, such as object classification, tracking, image and video segmentation, etc. But the main challenge of this field without a doubt is an object detection task. This chapter contains an overview of the evolution of object detection from traditional approaches, to two-stage and one-stage detectors, their advantages and disadvantages, mathematical structure and overall modern Deep Learning methods, as well as a review of hyperparameter optimization algorithms and the rationale behind the chosen optimization strategy.

1.1 Evolution of Object Detection

Two main object detection tasks are to classify and localize the object on the image. The whole evolution of this task is focused on creating approaches that will firstly be able to answer what object is on the image and where it's located, and secondly – do it as fast as possible. But the problem is that the complexity of the solution depends on the type of task at hand. Apart from basic computer vision challenges, such as different angles of the object, different lighting and the variety of images in one class, the detection task demands finding the object and to qualitatively localise it when the rotation and scale of it have been changed, or it's partly covered, etc [1].

Chronology of object detection splits into 2 parts: before 2014 – traditional approaches, and after 2014 – deep learning era. The issue with lack of images led to creation of very complicated handcrafted algorithms to extract the features out of the image, this became the main problem of traditional approaches. They were multi-stage, composed of separate algorithms and models with no ability to train their full pipeline.The following section outlines the key milestones of this era and their role in shaping modern detection.

1.1.1 Traditional Approaches

The first notable object detection approach was the Viola-Jones detector, designed specifically for face detection. It applied a sliding window across the entire image to determine whether a given region contained a face. The method was highly constrained: it required good lighting conditions, frontal face orientation, and the absence of occlusions, which made the detector sensitive and brittle to any variation in input conditions.

A more general approach followed with the combination of Histogram of Oriented Gradients (HOG) and Support Vector Machine (SVM). HOG is a feature descriptor that represents image regions as vectors of gradient information. It operates by rescaling the image and applying a sliding window, within which local intensity gradients are computed — regions with sharp changes in color or brightness receive higher values, and the direction of the change is also recorded. The resulting feature vector captures the local shape and edge structure of the region, which is then passed to an SVM for classification.

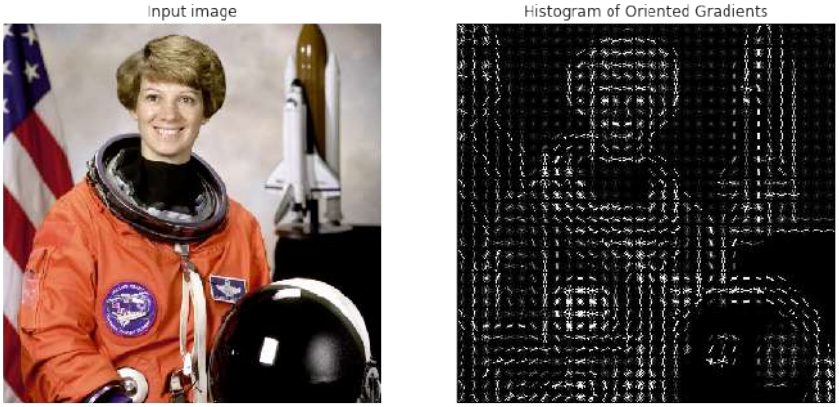


Figure 1.1: Example of HOG descriptor operation. Source: Medium

Support Vector Machine is a binary classification method that finds an optimal separating hyperplane between two classes by maximizing the margin between the nearest data points from each class [2]. Formally, the margin is defined as:

$$d = \frac{2}{\|w\|} \tag{1.1}$$

To maximize the margin, SVM minimizes the norm of the weight vector $\|w\|$:

$$\min_{w,b} \left(\frac{1}{2} \|w\|^2 \right) \tag{1.2}$$

subject to $y_i(w^T x_i + b) \geq 1, \quad \forall i.$

With the HOG descriptor, SVM can use the extracted features for object classification. However, this combination remains sensitive to changes in object pose — variations in human posture, for example, produce substantially different edge patterns, making it difficult for the model to generalize across different appearances of the same class.

The state-of-the-art among traditional approaches is the Deformable Part-based Model (DPM), which extended the HOG framework by treating objects as spatial compositions of their parts. Rather than detecting an object as a whole, DPM detects constituent parts independently — for a person, these might include the head, arms, and legs. If enough parts are detected in spatially consistent positions, the SVM classifies the overall region as a positive detection.

Although traditional methods established a strong conceptual foundation for object detection, they shared a common limitation: their multi-stage, handcrafted nature made them slow and difficult to scale, which motivated the shift towards end-to-end deep learning approaches [1].

1.1.2 Deep Learning era: Two-stage Detectors

Since 2014, object detection has evolved rapidly along two main directions. Two-stage detectors follow a "coarse-to-fine" approach: they first identify approximate regions likely to contain objects, then refine these proposals into precise localization predictions. One-stage detectors, by contrast, treat the initial region estimate as the final prediction, which makes them significantly faster but generally less precise in localization. This subsection focuses on the development of two-stage detectors[1].

The first two-stage detector proposed was Regions with Convolutional Neural Network (RCNN). Using a selective search algorithm, it extracts approximately 2000 region proposals that may contain objects and rescales each to a fixed size — a necessary step since CNN architectures require fixed-size inputs to produce feature maps of consistent dimensions.

To better understand the RCNN architecture, it is useful to first review the basic structure of a CNN. The network consists of five processing stages [3]:

1. **Input.** The network receives the image as a matrix of pixel values.

2. **Convolution layers.** Small matrices called filters (e.g. 3×3) slide across the image to extract local features. Each filter is sensitive to different patterns. The convolution operation multiplies each filter element by the corresponding image pixel and sums the results:

$$S_{i,j} = I_{i,j} K_{m,n} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I_{i+m,j+n} K_{m,n} \tag{1.3}$$

   - $I_{i,j}$ – pixel value at position (i, j) in the input image;
   - $K_{m,n}$ – weight value at position (m, n) of the filter;
   - $S_{i,j}$ – output feature map value at position (i, j);

- M – filter width, N – filter height.

3. **Activation.** A nonlinear function is applied to enable the network to model complex relationships:

$$a_{i,j} = \varphi(S_{i,j}) \tag{1.4}$$

- $a_{i,j}$ – activated nonlinear feature;
- φ – activation function.

The most widely used activation function is the Rectified Linear Unit (ReLU):

$$ReLU(S) = \max(0, S) \tag{1.5}$$

4. **Pooling layers.** These layers reduce the spatial dimensions of the feature map, decreasing the number of parameters and computations in subsequent layers. A sliding window of size k is applied with an aggregation operation. The most common variant is max pooling, which selects the maximum value within each window:

$$y_{i,j} = \max_{m,n \in \text{window}} (a_{i+m,j+n}) \tag{1.6}$$

- a – feature map;
- (i, j) – window position;
- (m, n) – local offset within the window.

5. **Fully-connected layers.** Each neuron in a fully-connected layer is connected to all neurons in the previous layer. The layer applies a linear transformation followed by an activation function:

$$z = Wx + b \tag{1.7}$$

- x – input vector;
- W – weight matrix, where $W_{i,j}$ is the weight of the connection between the j-th input and i-th output neuron;
- b – bias vector;
- z – output vector.

Once features are extracted by the CNN, an SVM classifier determines whether an object is present in the region and assigns a class label. RCNN demonstrated a notable improvement in detection quality: mAP increased from 33.7% achieved by DPM to 58%. However, the pipeline was extremely slow — processing a single image took approximately 14 seconds, largely due to redundant feature computation across approximately 2000 overlapping region proposals.

The next step was SPPNet — Spatial Pyramid Pooling Network — which addressed this inefficiency by computing a single feature map for the entire image rather than separately for each region proposal, reducing redundant computation by a factor of up to 20. The Spatial Pyramid Pooling layer also resolved the fixed-size input constraint of standard CNNs by producing a fixed-length feature vector regardless of the input image dimensions, enabling the network to process images of varying sizes.

All prior approaches operated as pipelines of independent components: selective search for region proposals, CNN for feature extraction, and SVM for classification, each trained separately. Fast RCNN proposed a unified network capable of jointly training the object classifier and bounding box regressor in a single end-to-end framework. This architectural change brought a substantial improvement in both speed (approximately 200 times faster than RCNN) and accuracy, with mAP reaching 70%.

Faster RCNN introduced the Region Proposal Network (RPN) — a subnetwork that integrates region proposal generation, feature extraction, and bounding box regression into a single trainable module. The RPN produces a set of candidate regions and scores them by their likelihood of containing an object, replacing the external selective search algorithm used in earlier approaches. While Faster RCNN reduced processing time further, some redundancy in computation remained.

In 2017, the Feature Pyramid Network (FPN) was introduced as a built-in module for multi-scale object detection. Deep layers of a CNN capture abstract semantic information but lose spatial resolution, making it harder to localize small objects. FPN addresses this by constructing a top-down feature pyramid: high-level semantic features from deeper layers are progressively combined with higher-resolution features from shallower layers, enabling accurate detection across a range of object scales. FPN subsequently became a standard component in high-performing two-stage detector architectures [1].

1.1.3 Deep Learning era: One-stage Detectors

Two-stage detectors showed great accuracy, however they remain complex and insufficiently fast. First, they generate region proposals, then clarify whether these regions contain any object and of which class, and define coordinates that best fit the object. Meanwhile, one-stage detectors are outstanding in speed by detecting objects in just one inference, however this approach affects the accuracy in detecting small and dense ob-

jects. This is due to the imbalance of the amount of background and object (foreground) instances [1].

You Only Look Once (YOLO) is a first one-stage detector [1]. It runs a single neural network on the image. The network splits the image into regions, detecting objects, bounding boxes and their probabilities simultaneously. This approach is significantly faster, however the model underperforms in object localization.

In 2015, the Single Shot Multibox Detector (SSD) was proposed [1]. The main difference of SSD is that it detects objects of different scales at different network layers, compared to previous detectors that detected objects only on the upper layers. This improved the accuracy, especially for small objects, although at a higher computational cost than YOLO.

The RetinaNet model solves the issue with imbalance of background and foreground instances by introducing the focal loss function (FL)[1]. FL is a modified binary cross-entropy function that penalizes the model more heavily for misclassifying hard instances. As a result, the model assigns less weight to easily classified examples and pays more attention to the actual objects. This approach showed significant mAP improvement that became competitive with two-stage detector performance.

CornerNet introduced an anchor-free detection paradigm, reformulating object detection as a keypoint estimation problem. This eliminated the need for manual anchor configuration and addressed the class imbalance issue inherent to anchor-based approaches. As a result, CornerNet surpassed the majority of one-stage detectors of its time, achieving a COCO mAP@.5 of 57.8%[1].

CenterNet, proposed in 2019, advanced the keypoint-based detection paradigm by representing each object as a single center point. This eliminated complex post-processing steps such as keypoint grouping and NMS, resulting in a fully end-to-end pipeline. The framework also proved easily extensible to auxiliary tasks including 3D object detection and pose estimation. Despite its conceptual simplicity, CenterNet achieved competitive results with a COCO mAP@.5 of 61.1% [1].

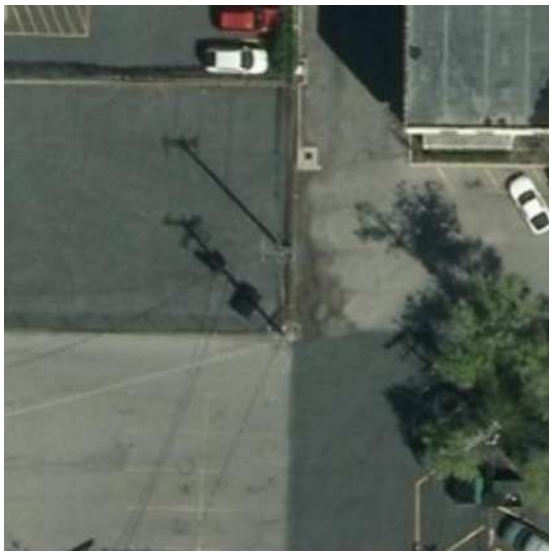
DETR, proposed in 2020, marked a fundamental shift in object detection by adopting a Transformer-based architecture and framing detection as a set prediction problem. This fully end-to-end approach removed the need for both anchor boxes and anchor points. However, DETR suffered from slow convergence and weak performance on small objects, which motivated the development of Deformable DETR — an improved variant that addressed these limitations and achieved state-of-the-art results with a COCO mAP@.5 of 71.9% [1].

Nevertheless, despite these architectural advances, YOLO-based models remain dominant in real-time detection scenarios due to their superior inference speed and beneficial accuracy-latency trade-off [1].

1.2 Detection in Aerial Imagery

Object detection in aerial imagery is fundamentally different from detecting objects in photos taken by handheld cameras, and cannot be directly approached the same way[4].

The most obvious difference is the viewing angle. Ground-level cameras capture objects from the side, so their shape and size stay relatively consistent. UAV cameras, however, shoot from above at varying altitudes and angles, which means the same object can look completely different depending on the shot. This forces the model to recognize far more visual variations of the same object class.



(a) Source: [5]

(b) Source: [6]

Figure 1.2: Example of UAV imagery

Another challenge is the size of the object. In regular photos, a car or a person takes up a noticeable part of the frame. In aerial imagery, the same objects may cover only a few pixels, losing most of the visual detail that detection models typically rely on. Handling this requires specialized architectural components, such as feature pyramid networks or attention mechanisms, to extract meaningful information from very small regions.

The image quality adds further difficulty. Drones are subject to vibration and rapid movement, which can cause blur and distortion. Lighting conditions also change more dramatically at altitude, and sensor resolution is generally lower than in ground-level photography.

Finally, occlusion works differently in aerial scenes. Instead of one object blocking another, targets are often hidden under trees, rooftops, or other environmental elements, which is a less predictable and harder pattern for a model to learn.

All of these factors make aerial imagery a distinct problem that requires its own dedicated solutions[4].

1.3 Hyperparameter Optimization Algorithms

This chapter presents an analysis of algorithms for hyperparameter optimization in order to select the most suitable approach for the task. As established in the introduction, hyperparameter selection is a separate optimization problem that directly affects model performance. The following subsections review five commonly used HPO strategies and motivate the final choice.

1.3.1 Grid Search

Grid search is a straightforward approach to hyperparameter optimization that exhaustively evaluates all possible combinations of predefined values [7]. Formally, it evaluates the objective function over the Cartesian product of discrete sets of values for each hyperparameter:

$$x^* = \arg\min_{x \in \mathcal{X}} f(x), \quad \mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n \tag{1.8}$$

$\mathcal{X}_i$ — discrete set of values for hyperparameter i;

$f(x)$ — objective function.

This approach may be suitable in cases where the evaluation cost is negligible. However, grid search scales poorly with dimensionality: the number of evaluations grows exponentially with the number of hyperparameters. Each evaluation of $f(x)$ corresponds to a full model training run, which typically takes hours. For example, with 5 hyperparameters and 3 candidate values each, grid search requires $3^5 = 243$ evaluations. Assuming each training run takes approximately 4 hours on a GPU, this amounts to roughly 40 days of continuous computation. Beyond the computational cost, grid search allocates equal resources to all combinations, including those in clearly unpromising regions of the space, leaving no room for learning from previous evaluations [8]. For these reasons, grid search was rejected as a viable option for this task.

1.3.2 Random Search

Random search is an uninformed search method that samples hyperparameter configurations independently from previous evaluations. Each set is drawn from a probability distribution over the hyperparameter space, evaluated, and then a new independent set is sampled:

$$x_i \sim p(\mathcal{X}) \tag{1.9}$$

x_i — i-th sampled hyperparameter configuration;

$p(\mathcal{X})$ — probability distribution over the hyperparameter space $\mathcal{X}$.

In practice, most hyperparameter optimization problems have low effective dimensionality, meaning that only a subset of hyperparameters has a meaningful impact on the objective function [8]. This property makes random search more efficient than grid search for a fixed evaluation budget, as random sampling is more likely to cover the important dimensions adequately. The method is also significantly cheaper computationally, since the number of evaluations is fixed rather than growing exponentially.

However, random search is not a systematic method [8]. Since each sample is drawn independently, the algorithm does not learn from past evaluations and may miss configurations in promising regions of the search space. Despite this limitation, random search serves as an effective initialization strategy: it can quickly explore broad areas of the hyperparameter space and provide a set of informative starting points for a more targeted algorithm. For this reason, random search was selected as the initialization stage in the two-stage HPO pipeline used in this work.

1.3.3 Bayesian Optimization

Bayesian optimization is a general framework for solving optimization problems where the objective function is expensive to evaluate [9]. Unlike random search, which samples configurations independently, Bayesian optimization uses information from previous evaluations to guide the search.

The setup is as follows: given an objective function $f(x)$, where x is a hyperparameter configuration and $f(x)$ is the resulting model performance metric, the goal is to find x^* that maximizes $f(x)$. The key difficulty is that this optimization problem is not differentiable in the traditional sense — unlike training a neural network, where gradients can be computed with respect to model weights, there is no closed-form expression linking hyperparameters to model accuracy. The only way to evaluate a given configuration is to run a full training cycle and measure the result.

Bayesian optimization addresses this by constructing a probabilistic surrogate model that, based on previously evaluated points, predicts the expected value of the function at a new candidate point x^*. This makes it well-suited for hyperparameter optimization, as each new trial is informed by all prior trials, allowing the algorithm to focus the search on the most promising regions of the space. Several specific algorithms implement this general strategy, two of which are reviewed below.

1.3.4 Gaussian Process

The Gaussian Process (GP) is one realization of the Bayesian optimization framework. It constructs a probabilistic model over the objective function by assuming that points

close to each other in the parameter space are likely to have similar function values. This similarity is quantified using a kernel function $K(x_i, x_j)$, which measures how strongly two points x_i and x_j influence each other. For all observed pairs, a covariance matrix K is constructed [10]:

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & \dots & x_n \end{array} \\ \begin{bmatrix} k_{11} & k_{12} & k_{13} & \dots & k_{1n} \\ k_{21} & k_{22} & k_{23} & \dots & k_{2n} \\ k_{31} & k_{32} & k_{33} & \dots & k_{3n} \\ & & \dots & & \\ k_{n1} & k_{n2} & k_{n3} & \dots & k_{nn} \end{bmatrix} \end{array} \tag{1.10}$$

This covariance matrix describes the mutual similarity between all observed points. To predict the function value at a new point x^*, the standard Gaussian process prediction formula is applied [10]:

$$\mu(x^*) = k_*^T \cdot K^{-1} \cdot y \tag{1.11}$$

where

k_* — covariance vector between the new point x^* and all previously observed points x_i;

y — vector of observed function values $y_1, \dots, y_n$;

K^{-1} — inverse of the covariance matrix K.

Despite being a well-established method, the Gaussian Process has several practical limitations in the context of this task:

1. It requires computing the inverse of the covariance matrix, which is computationally expensive and scales cubically with the number of observations;
2. It can only handle continuous variables natively, while hyperparameters in detection models may be categorical or discrete;
3. Its computational cost increases with each additional evaluation, making it progressively slower over the course of a search;
4. It does not scale well to high-dimensional hyperparameter spaces.

For these reasons, the Gaussian Process was not selected. However, the Bayesian optimization framework itself was identified as a suitable strategy, and further analysis focused on finding a more practical algorithm within this approach.

1.3.5 Tree-Structured Parzen Estimator

Tree-structured Parzen Estimator (TPE) is an algorithm based on Bayesian optimization that also constructs a probabilistic model, but uses a different and computationally simpler approach to model the objective function [9]. Instead of modeling $p(y|x)$ directly, TPE models $p(x|y)$ — the distribution of hyperparameters conditioned on the quality of the result.

First, the algorithm divides all previously evaluated hyperparameter configurations into two groups: those that produced results above a certain quality threshold ("good"), and those that produced results below it ("bad"). The threshold is defined as:

$$p(y < y^*) = \gamma \tag{1.12}$$

where

- γ — a hyperparameter that determines the proportion of configurations considered "good". For example, if $\gamma = 0.25$, the algorithm takes the top 25% of evaluated configurations as the good set;
- y^* — the threshold value separating good from bad results.

The conditional distribution of hyperparameters is then defined as:

$$p(x|y) = \begin{cases} \ell(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases} \tag{1.13}$$

where

- y — result of the objective function for a given configuration;
- y^* — the quality threshold;
- $\ell(x)$ — density estimate built from configurations that outperformed the threshold;
- $g(x)$ — density estimate built from configurations that did not reach the threshold.

Based on these two subsets, the algorithm builds separate statistical distribution models ($\ell(x)$ and $g(x)$) for each hyperparameter. An important practical property of TPE is its ability to handle different variable types: for continuous numerical parameters it fits normal, log-uniform, or mixture-of-Gaussians distributions; for categorical parameters it uses a weighted categorical distribution where weights reflect how frequently each value appears in the good or bad subset; and for discrete parameters it applies a discrete distribution.

To illustrate: for the learning rate parameter, the algorithm builds a distribution of its values separately in the good and bad subsets. Since learning rate is a continuous

variable, this will typically be a normal distribution where the peak corresponds to the value most frequently associated with good results.

Each parameter has its own distribution in both subsets. To evaluate a full hyperparameter configuration rather than individual parameters, TPE assumes independence between parameters and multiplies their individual densities to obtain joint densities $\ell(x)$ and $g(x)$ for the full set. The algorithm then computes their ratio:

$$\frac{\ell(x)}{g(x)} \tag{1.14}$$

A higher ratio indicates that the configuration is more likely to belong to the good subset.

To decide which configuration to evaluate next, TPE uses the Expected Improvement (EI) criterion — a standard Bayesian optimization acquisition function that estimates how much improvement a candidate configuration is likely to bring over the current threshold. This prevents the search from getting stuck in a narrow region of similar well-performing configurations. On each iteration, TPE generates multiple candidate configurations from the $\ell(x)$ distribution and uses EI to select the most promising one. The EI formula is:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y)p(y|x)dy \tag{1.15}$$

where

- y — possible result;
- $p(y|x)$ — the probability of obtaining result y with configuration x;
- $(y^* - y)$ — the magnitude of improvement over the threshold.

Since $p(y|x)$ is not directly available, Bayes' theorem is applied to express it through the quantities the algorithm does model [9]:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1.16}$$

Which gives:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{1.17}$$

Substituting this into the EI integral:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y)\frac{p(x|y)p(y)}{p(x)}dy$$

From equation 1.12, $\gamma = p(y < y^*)$, and the marginal $p(x)$ can be expressed as: $p(x) = \int_{\mathbb{R}} p(x|y)p(y)dy = \gamma\ell(x) + (1-\gamma)g(x)$.

Therefore:

$$\int_{-\infty}^{y^*}(y^*-y)p(x|y)p(y)dy = \ell(x)\int_{-\infty}^{y^*}(y^*-y)p(y)dy = \gamma y^*\ell(x) - \ell(x)\int_{-\infty}^{y^*}p(y)dy$$

Which yields:

$$EI_{y^*}(x) = \frac{\gamma y^*\ell(x) - \ell(x)\int_{-\infty}^{y^*}p(y)dy}{\gamma\ell(x) + (1-\gamma)g(x)} \propto \left(\gamma + \frac{g(x)}{\ell(x)}(1-\gamma)\right)^{-1}$$

This result shows that maximizing EI comes down to selecting configurations with high $\ell(x)$ and low $g(x)$ probability. The algorithm favors configurations that resemble successful trials and avoids those similar to unsuccessful ones. Among all generated candidates, the one with the highest EI value is selected for the next evaluation.

The TPE algorithm has several practical advantages over the previously reviewed methods. It samples new candidates from the $\ell(x)$ distribution directly, which is fast and computationally cheap compared to Gaussian process-based Bayesian optimization, especially when the search space is large and contains parameters of mixed types. Unlike random search and grid search, TPE learns from each completed trial, progressively focusing the search on more promising regions. For these reasons, TPE was selected as the primary optimization algorithm for this work.

1.3.6 Genetic Algorithm

The Ultralytics framework provides a built-in mechanism for automatic hyperparameter optimization based on a genetic algorithm. The approach imitates natural selection: an initial set of hyperparameter combinations (learning rate, momentum, augmentation parameters, etc.) is evaluated by training the model with each configuration. Combinations that got the best mAP scores are retained, and a mutation operator is applied to them — introducing small random changes to the parameter values. This cycle repeats over multiple iterations, gradually narrowing the search space toward an optimal configuration[11]. By default, the algorithm runs for 300 iterations, with mutation strength decreasing linearly from 0.2 to 0.1, which makes the search more precise over time. The genetic algorithm was not used in this study due to computational constraints. Since it requires training the model hundreds of times with different configurations, the process demands significant GPU resources and time. Given the limited hardware available, this approach was impractical. The default Ultralytics hyperparameters provided sufficient performance for the task.

Chapter 2

Detection Model Selection

This chapter provides an overview of the base model selected for the detection task, a description of its architecture, and the reasoning behind the choice.

2.1 Motivation

The base model selected for this work is YOLOv8 Nano — a lightweight, one-stage, anchor-free detector. The selection was driven by the requirements of real-time detection: the model must be capable of processing frames within milliseconds and returning results with minimal latency, while also being lightweight enough for potential deployment on resource-constrained hardware. There is a well-known trade-off between detection accuracy and inference speed — given more processing time, a model can generally produce better results — and the goal was to find a practical balance between the two.

Ultralytics, the computer vision platform behind the YOLO family of detectors, positions their models specifically as real-time object detectors. This positioning is supported by practical benchmarks: YOLO models offer the best compromise between speed, model size, and detection quality compared to alternatives such as DETR-based detectors. While DETRs (DEtection TRansformers) achieve strong accuracy, they historically have higher computational complexity and longer training times, making them less suitable for real-time applications where low-latency inference is a priority. YOLO models, by contrast, prioritize speed while maintaining competitive accuracy.

Table 2.1: Performance comparison of selected object detection models.

Model	**mAP$^{\text{val}}_{50\text{-}95}$**	**Speed (ms)**	**Params (M)**	**FLOPs (B)**
YOLOv8n	37.3	1.47	3.2	8.7
YOLOv8s	44.9	2.66	11.2	28.6
YOLOv8m	50.2	5.86	25.9	78.9
RTDETRv2-s	48.1	5.03	20.0	60.0

Speed measured on T4 GPU with TensorRT. Source: Ultralytics Documentation

Previous YOLO versions are anchor-based, meaning they rely on predefined bounding boxes of various sizes and aspect ratios to localize objects. YOLOv8 is the first anchor-free model in the YOLO family — it directly predicts the center coordinates, width, and height of the object along with the class probability, without depending on preset anchor configurations. This simplifies the computation, reduces inference time, and removes the need for manual anchor tuning.

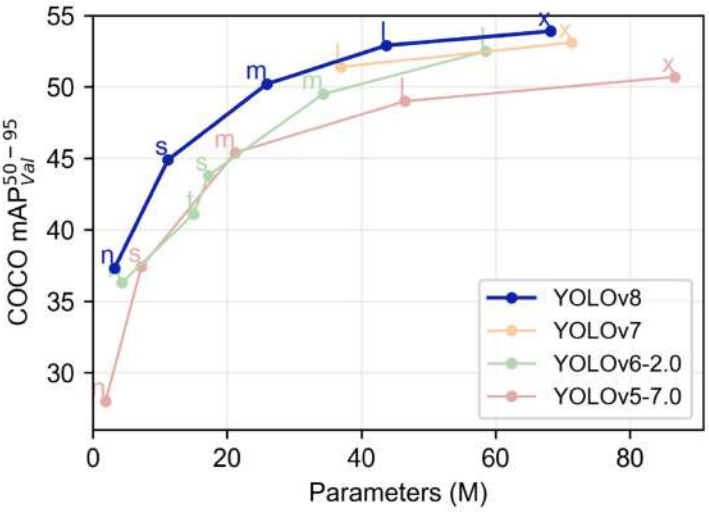


Figure 2.1: YOLO performance comparison. Source: MMYOLO

For real-time detection on a UAV, the model must operate at interactive frame rates (approximately 30 FPS or higher), as any delay in processing directly affects the responsiveness of the system. Among the YOLOv8 model variants, the Nano version is the lightest, with approximately 3.2 million parameters, while still maintaining sufficient accuracy for the task.

Additionally, the model provides pre-trained weights obtained from training on the COCO (Common Objects in Context) dataset. This transfer learning approach allows the model to start with a general understanding of visual features and object structures, rather than learning from scratch with random weight initialization. Fine-tuning from pre-trained weights typically leads to faster convergence and better results, especially when the custom dataset is relatively small.

2.2 Model's Architecture

The YOLOv8 architecture consists of three main components: Backbone, Neck, and Head [12], as shown in Figure 2.2.

Backbone. The backbone is a CNN-based feature extractor. It takes an image as input and produces feature maps at different scales, capturing patterns ranging from simple textures and edges to more abstract semantic concepts. Compared to previous versions,

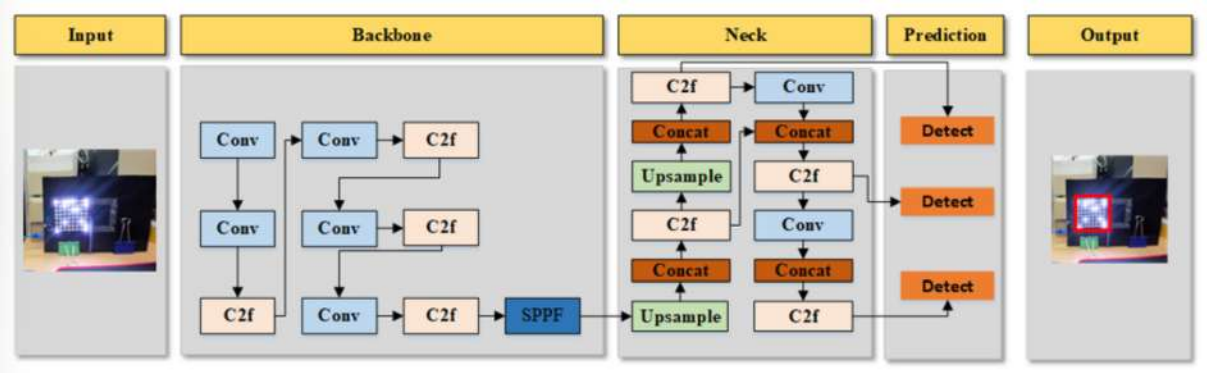


Figure 2.2: YOLOv8 Architecture. Source: [12]

YOLOv8 replaces the C3 modules with C2f (Cross Stage Partial with 2 convolutions and additional feature flow) [13]. The C2f module improves accuracy by combining high-level features with contextual information. It receives a feature map as input and splits it into two parts: one part passes directly to the concatenation step, while the other goes through N bottleneck blocks. Each bottleneck compresses the feature map channels using a 1×1 convolution, processes them with a 3×3 convolution, and restores the original channel count. All bottleneck outputs are then concatenated with the first part of the feature map, and a final convolution layer adjusts the result to the required dimensions. At the end of the backbone, a Spatial Pyramid Pooling Fast (SPPF) module is applied. SPPF enriches the final feature map by pooling it at multiple scales, providing the neck with a richer contextual representation before feature fusion begins. The backbone outputs three feature maps at different scales:

- **P3** $(80 \times 80 \times 256)$ — high spatial detail, limited semantic information;
- **P4** $(40 \times 40 \times 512)$ — balanced level of spatial detail and semantic features;
- **P5** $(20 \times 20 \times 512)$ — low spatial detail, rich semantic information.

Neck. The neck is responsible for multi-scale feature fusion. It receives the three feature maps from the backbone at different spatial resolutions and combines them in two passes. In the first pass, semantic information from deeper layers is propagated downward to enrich shallower, more spatially detailed feature maps. In the second pass, spatial details are passed back upward. This is achieved by taking deeper feature maps, merging them with shallower ones, and processing the result through C2f blocks. As a result, each output feature map contains both high-level scene understanding and low-level spatial detail, which helps the model detect objects of varying sizes[12].

Head. The head is responsible for the final detection, localization, and classification of objects. Previous YOLO versions used a coupled head, where bounding box regression and class prediction were performed on the same layers. YOLOv8 introduces a decoupled

head with two separate branches for these tasks. Since defining a bounding box and classifying an object are fundamentally different problems, dedicated branches allow each to specialize independently, improving overall performance. Additionally, YOLOv8 uses the anchor-free approach described in the previous section, predicting bounding box coordinates directly without relying on predefined anchor boxes. This simplifies the training process and reduces the number of configuration choices that need to be made manually[12].

2.3 Loss Function

The loss function is a formula used to measure the difference between the model's prediction and the ground truth. It assigns a penalty score when the prediction is incorrect, and the model must minimize this score during training to improve its accuracy [14]. **Minimizing this loss function is the internal optimization process that occurs during model training.**

YOLOv8 uses a combination of three loss functions — bounding box loss, classification loss, and distribution focal loss — each measuring a different aspect of the model's output. The total loss is computed as a weighted sum:

$$\mathcal{L} = \lambda_{box}\mathcal{L}_{box} + \lambda_{cls}\mathcal{L}_{cls} + \lambda_{dfl}\mathcal{L}_{dfl} \tag{2.1}$$

where

- λ_i — the weight of each loss component;
- $\mathcal{L}_i$ — the value of each loss component.

The default weight values in the YOLOv8 configuration are $\lambda_{box} = 7.5$, $\lambda_{cls} = 0.5$, and $\lambda_{dfl} = 1.5$. The following subsections describe each of the three loss functions.

2.3.1 Bounding Box Loss

The bounding box loss measures how accurately the model predicts the location and shape of the bounding box around an object. YOLOv8 uses the Complete Intersection over Union (CIoU) function for this purpose:

$$\mathcal{L}_{box} = 1 - \text{IoU} + \frac{\rho^2(b, b^{gt})}{c^2} + \alpha v \tag{2.2}$$

where

- $\rho^2(b, b^{gt})$ — the squared Euclidean distance between the centers of the predicted and ground-truth boxes;

- c — the length of the diagonal of the smallest enclosing box that covers both the predicted and ground-truth boxes;
- α — a balancing parameter;
- v — a term that measures the consistency of the aspect ratio between the two boxes.

The IoU component is the ratio of the intersection area to the union area of the predicted and ground-truth boxes:

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{2.3}$$

CIoU extends the basic IoU by additionally penalizing the distance between box centers and differences in aspect ratio, which helps the model converge faster when the predicted box is far from the ground truth.

2.3.2 Classification Loss

The classification loss measures how well the model predicts the correct class label for each detected object. YOLOv8 uses Binary Cross-Entropy (BCE) for this purpose [15]:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i)) \tag{2.4}$$

where

- y — the ground-truth label (0 or 1 for each class);
- $p(y)$ — the predicted probability that the object belongs to the given class;
- N — the total number of objects.

YOLOv8 applies independent sigmoid activations, meaning the BCE function is computed for each class separately as its own binary classification problem. This allows the model to handle multi-class detection by treating each class independently rather than as a single mutually exclusive decision.

2.3.3 Distribution Focal Loss

The third component is the Distribution Focal Loss (DFL), which helps the model predict bounding box coordinates more precisely. Instead of predicting exact coordinate values directly, DFL models the box edge positions as probability distributions over a set of discrete bins [16]. This approach is particularly useful when there is ambiguity in the exact object boundary, as it allows the model to express uncertainty about the position rather than committing to a single value.

$$\mathrm{DFL}(\mathcal{S}_i, \mathcal{S}_{i+1}) = -\left[(y_{i+1} - y)\log(\mathcal{S}_i) + (y - y_i)\log(\mathcal{S}_{i+1})\right] \tag{2.5}$$

where

- y — the ground-truth coordinate value;
- y_i — the nearest bin to the left of the ground truth ($y_i \leq y$);
- $\mathcal{S}_i$ — the predicted probability that the bounding box edge falls in the y_i bin;
- $y - y_i$ — the distance from the ground-truth value to the neighboring bin.

The final predicted edge coordinate is computed as the weighted average across all bin positions:

$$\hat{y} = \sum_{i=0}^{n} i \cdot \mathcal{S}_i \tag{2.6}$$

The weights λ_{box}, λ_{cls}, and λ_{dfl} that control the contribution of each loss component are among the hyperparameters optimized in this work, as their values directly affect the training dynamics and the resulting model performance.

Chapter 3

Approach

This chapter describes the practical work carried out in this project. It begins with the formal problem formulation, followed by an overview of the training pipeline and the data preparation stages. The core of the chapter covers the hyperparameter optimization process — the search space definition, the two-stage optimization strategy, and its implementation. Finally, the evaluation methodology is described, including the metrics used and the statistical testing approach.

3.1 Problem Formulation

Training an object detection model involves two levels of optimization. The inner level is the training process itself: given a training dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$, where x_i is an input image and y_i is the corresponding ground-truth annotation, the model learns a set of internal weights θ by minimizing the composite loss function:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i) \tag{3.1}$$

where $f_\theta(x_i)$ is the model's prediction for input x_i, and $\mathcal{L}$ is the composite loss function described in Chapter 2, consisting of bounding box, classification, and distribution focal loss components. This process is handled by the training algorithm and is not controlled directly.

The outer level — and the focus of this work — is hyperparameter optimization. Hyperparameters λ, such as learning rate, momentum, and loss component weights, are set before training and directly influence how the inner optimization proceeds. The goal is to find the configuration λ^* that yields the best model performance on a validation set:

$$\lambda^* = \arg\max_{\lambda \in \Lambda} \mathrm{mAP}_{50}(f_{\theta^*(\lambda)}, D_{val}) \tag{3.2}$$

where Λ is the hyperparameter search space, $\theta^*(\lambda)$ denotes the optimal model weights

obtained by training with configuration λ, and mAP_{50} is the mean Average Precision at IoU threshold 0.5, evaluated on the validation set D_{val}. This metric is described in detail in Chapter 4.

These two levels are nested: each evaluation of a candidate hyperparameter configuration requires a full training run, making the outer optimization computationally expensive [9]. The approach taken in this work is a two-stage hyperparameter search algorithm designed to explore the space Λ efficiently within a feasible computational budget.

3.2 Pipeline Setup

This section describes the data-to-model workflow used in this project. The pipeline consists of seven stages, progressing from raw data through preprocessing, training, and evaluation. It was developed on top of an existing codebase. Several stages were redesigned, validation checks were introduced at multiple points, the hyperparameter optimization workflow was implemented, and statistical significance testing was added for the evaluation results.

3.2.1 Dataset Overview

The dataset used in this work consists of 20,209 images. It covers 10 object classes relevant to the target detection scenario, such as military vehicles, aircrafts and civilian objects. The class distribution across the dataset is notably imbalanced, with some categories such as tank_spg being significantly more represented than others such as ew_radar or aircraft.

Table 3.1: Class distribution in the dataset

Class	**Annotations**
tank_spg	8176
apc	2111
civilian_vehicle	1690
military_person	1619
armored_vehicle	1432
civilian_person	1142
military_truck	1370
artillery	750
aircraft	132
ew_radar	39
Total annotations	**18,461**

The dataset also includes a portion of background images containing no annotated objects. It is important given that the model will be run on continuous UAV footage

where most frames have no objects. Including such negative samples during training helps the model avoid false positive detections on empty frames, which directly improves precision in real-world deployment.

3.2.2 Stage-by-Stage Description

Stage 0 – Metadata Preparation. Images from selected UAV datasets are stored in cloud storage. Metadata covering image identifiers, source datasets, and class label assignments is maintained in a versioned spreadsheet and retrieved via API at the start of each run. Then it is merged with filesystem records into a single CSV file that is used for all subsequent filtering. A filtering column in the configuration spreadsheet controls which datasets are included in training, without the need to modify the code.

Stage 1 – Duplicate Detection and Data Selection. Perceptual hashing is applied to every image, and pairwise Hamming distances are computed across the full set. Images below a minimum distance threshold are treated as duplicates and removed. This is important for preventing visually redundant samples from appearing across the train, validation, and test splits.

Stage 2 – Class Remapping. Source datasets use inconsistent class naming conventions. This stage normalizes all annotation labels to a unified taxonomy defined in the same versioned configuration document. The original mapping logic was replaced with a configuration-driven approach, and a validation check was added: any remapped label without a matching class definition raises an error at preprocessing time rather than causing issues during training.

Stage 3 – Dataset Splitting. The filtered dataset is split into training, validation, and test subsets in an 80/10/10 ratio using stratified sampling to preserve per-class proportions. A class distribution report is generated so the split balance can be verified before training begins.

Stage 4 – Augmentation. A custom augmentation pipeline is applied during training to simulate common UAV image degradation: downscaling, JPEG compression, shot noise, and Gaussian blur.

Stage 5 – Model Training and Hyperparameter Optimization. The model is trained using a YOLOv8n architecture with Stochastic Gradient Descent optimizer. Hyperparameter selection is carried out through a two-phase search combining random exploration and TPE-based optimization, implemented with Optuna. This process is described in detail in Section 3.3.

Stage 6 – Evaluation. Evaluation is conducted in two steps: automated metric evaluation and human evaluation. In the first step, model performance is measured on the held-out test set using per-class AP@0.5, with a Wilcoxon signed-rank test applied to assess whether the HPO-tuned model improves over the baseline in a statistically

meaningful way. The second step involves human evaluation as an additional reference point. Both steps are described in detail in Chapter 3.4–3.6.

3.3 Hyperparameter Optimization

This section describes the hyperparameter optimization pipeline, covering the search space definition, the two-stage optimization strategy, and the implementation details.

3.3.1 Search Space

The search space was defined to include seven hyperparameters that have the most direct impact on training dynamics and model performance:

Table 3.2: Hyperparameters and their descriptions.

Name	Default value	Search interval	Description
lr0	0.01	$[10^{-5}, 0.1]$	Initial learning rate, determining the step size for weight updates during training.
lrf	0.01	[0.01, 1.0]	Final learning rate as a fraction of the initial rate (lr0 $\times$ lrf), applied via a scheduler to anneal the learning rate over training.
momentum	0.937	[0.6, 0.98]	Momentum coefficient for the SGD optimizer, governing the contribution of previous gradient updates to the current parameter update.
weight_decay	0.0005	[0.0, 0.001]	L2 regularization coefficient that penalizes large weight magnitudes to mitigate overfitting.
box	7.5	[0.5, 10.0]	Weighting factor for the bounding box regression loss component within the total loss function.
cls	0.5	[0.1, 5.0]	Weighting factor for the classification loss component, determining its relative contribution to the total loss.
dfl	1.5	[0.5, 5.0]	Weighting factor for the distribution focal loss component, responsible for fine-grained bounding box coordinate regression.

Source: Ultralytics Documentation

These seven hyperparameters were selected because they have the most significant effect on model performance. The search ranges for each parameter were based on the bounds used in the Ultralytics built-in genetic algorithm[11]. Including too many parameters in the search space increases its dimensionality, which in turn raises the number of trials needed and the overall computational cost. For this reason, the search was limited to the most impactful configuration variables.

3.3.2 Optimization Pipeline Structure

The optimization pipeline combines two algorithms: random search and the Tree-structured Parzen Estimator (TPE), applied sequentially in two phases.

Algorithm 1: Tree-structured Parzen Estimator (TPE) with Random Search Initialization

Parameters: $N_{\text{init}} = 10$ (Random Search trials), $N_{\text{TPE}} = 10$ (TPE optimization trials), $N_s = 24$ (candidates for acquisition function).

$\mathcal{D} \leftarrow \emptyset$

`// Phase 1: Random Search Initialization`

for $n = 1, 2, \ldots, 10$ **do**

  Randomly sample $\boldsymbol{x}_n$ from search space

  $y_n := f(\boldsymbol{x}_n) + \epsilon_n$ `// Evaluate objective function`

  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_n, y_n)\}$

end

`// Phase 2: TPE Optimization`

for $n = 11, 12, \ldots, 20$ **do**

  Compute $\gamma \leftarrow \Gamma(N)$ with $N := |\mathcal{D}|$ `// Splitting data`

  Split $\mathcal{D}$ into $\mathcal{D}^{(l)}$ and $\mathcal{D}^{(g)}$ by quantile γ

  Build KDE models $p(\boldsymbol{x}|\mathcal{D}^{(l)})$ and $p(\boldsymbol{x}|\mathcal{D}^{(g)})$

  Sample $\mathcal{S} := \{\boldsymbol{x}_s\}_{s=1}^{24} \sim p(\boldsymbol{x}|\mathcal{D}^{(l)})$

  Pick $\boldsymbol{x}_n := \text{argmax}_{\boldsymbol{x} \in \mathcal{S}} \frac{p(\boldsymbol{x}|\mathcal{D}^{(l)})}{p(\boldsymbol{x}|\mathcal{D}^{(g)})}$ `// Maximize Expected Improvement`

  $y_n := f(\boldsymbol{x}_n) + \epsilon_n$ `// Evaluate objective function`

  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\boldsymbol{x}_n, y_n)\}$

end

$\boldsymbol{x}^* := \text{argmax}_{(\boldsymbol{x},y) \in \mathcal{D}}\ y$ `// Select best configuration`

Source: Adapted from [17]

The first phase uses random search as the initialization stage. Since TPE requires an initial set of evaluated configurations to construct its first probability density estimates, random search is used to build a database of hyperparameter configurations and their corresponding objective function results (mAP@0.5). This provides a broad coverage of the search space without any prior assumptions about which regions are promising.

In the second phase, TPE takes over and performs informed optimization based on the results collected during the random search. It uses Kernel Density Estimation (KDE) to model

the distribution of hyperparameter values separately for well-performing and poorly-performing configurations, and then samples new candidates from the regions that are more likely to yield higher mAP values.

This two-stage approach moves from exploration to exploitation: the random search phase covers a wide range of values across the search space, and the TPE phase then focuses on the most promising regions identified from those initial results. While TPE can generate initial random samples internally, explicitly separating the two phases provides more control over the exploration budget and makes the process more transparent. Compared to running either method alone, this combination provides a better balance between search coverage and computational efficiency.

3.4 Evaluation Metrics

The evaluation is needed to define whether the HPO algorithm and the pipeline changes led to a meaningful improvement, and how significant that improvement was. The evaluation consists of two stages:

1. **Metrics evaluation** — quantitative analysis using standard detection metrics on the held-out test dataset;
2. **Human evaluation** — qualitative review of model inference on image and video data.

The following subsections describe the metrics used for quantitative evaluation.

Precision and Recall. Precision measures how many of the objects predicted by the model as a given class are actually correct [18]:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{3.3}$$

where

- True Positives — correctly detected instances of the class;
- False Positives — instances that the model incorrectly assigned to the class.

Recall measures how many of the actual ground-truth objects the model was able to detect [18]:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3.4}$$

where

- False Negatives — ground-truth instances that the model failed to detect.

Every prediction returned by the model has a confidence score representing how certain the model is about that specific detection. For evaluation, predictions for each class are sorted by confidence from highest to lowest.

Mean Average Precision. Average Precision (AP) is defined as the area under the precision-recall curve. Like precision and recall, AP takes values between 0 and 1. In practice, AP is typically computed on a smoothed (interpolated) version of the curve [19]:

$$AP = \int_0^1 p(r)\, dr \tag{3.5}$$

where

- r — recall;
- $p(r)$ — precision at a given recall level r.

Mean Average Precision (mAP) is the primary metric used in object detection to evaluate how well the model detects and classifies objects simultaneously. It is computed as the mean of AP values across all classes:

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{3.6}$$

The mAP metric reflects the trade-off between precision and recall — improving one typically comes at the cost of the other. In this work, mAP@0.5 is used: a prediction is considered correct when the predicted bounding box overlaps the ground-truth bounding box by at least 50%, as measured by Intersection over Union.

F1 Score. Unlike AP, which summarizes performance across all confidence thresholds, the F1 score is computed at a specific confidence threshold. It is the harmonic mean of precision and recall at that point:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.7}$$

The F1 score is useful because neither precision nor recall alone gives a complete picture of model performance. A model may have high precision but low recall, or vice versa. The F1 score identifies the confidence threshold at which precision and recall are most balanced.

3.5 Statistical Testing

To verify that the performance improvement achieved through hyperparameter optimization is not due to random variation, a statistical significance test was conducted. The comparison was performed on per-class AP@0.5 values from the validation set, treating each class as a paired observation between the baseline and the tuned model.

The Wilcoxon signed-rank test was selected for this comparison, as the small sample size (10 classes) and non-normal distribution of differences require a non-parametric paired test. The test was run as one-sided, with the alternative hypothesis that the tuned model performs better than the baseline.

Cohen's d was computed to measure the effect size, and a 95% bootstrap confidence interval for the mean AP@0.5 difference was calculated to estimate the range of expected improvement.

3.6 Human Evaluation

While quantitative metrics provide a standardized way to measure model performance, they do not always reflect how a model behaves on real data. Human evaluation is a crucial step to evaluate whether the model is suitable for practical deployment.

The evaluation was performed by running model inference on a set of images and video sequences and reviewing the results manually. The focus was on the quality of object detection, localization, and classification — specifically, whether the model consistently finds objects, places bounding boxes accurately, and assigns correct class labels.

Figure 3.1: Model inference on UAV footage: a civilian person detected with 0.91 confidence.

No formal scoring scale was used; the goal was to form a qualitative judgment of whether the model could realistically be considered for production use. An example of accurate object localization and classification is shown in Figure 3.1.

Chapter 4

Evaluation

The following chapter presents the evaluation of the proposed approach. It begins with an analysis of the best-performing trial identified during hyperparameter optimization, followed by a comparison of model performance across key metrics. Finally, statistical significance testing is applied to validate that the observed improvements are reliable and not a result of random variation.

4.1 Best Trial

The hyperparameter optimization was conducted in two phases: 10 trials using random search followed by 10 trials using the TPE sampler, for a total of 20 trials. The top-3 results are summarized in Table 4.1.

Table 4.1: Top-3 trials performance metrics.

Trial	**mAP@0.5**	**Precision**	**Recall**
014	0.6212	0.7780	0.5241
005	0.6211	0.6686	0.6008
017	0.6167	0.7103	0.5831

The best-performing configuration was found in trial_014, achieving a mAP@0.5 of 0.6212. Among the top-3 trials, it showed the highest precision (0.7780), indicating that the model was relatively conservative in its predictions — when it detected an object, it was correct most of the time. However, the recall of 0.5241 indicates that a portion of objects were missed.

The hyperparameters from trial_014 were then used to train the final model for 50 epochs, the results of which are presented in the following section.

4.2 Performance Comparison

Finally, the results of evaluation of the baseline model and the model after applying the hyperparameters tuning. Both metrics were evaluated using the same test image set and metrics

described above: Precision, Recall, mAP@0.5, and F1-score. The recall metric the value corresponds to the confidence level = 0.0 to show the maximum coverage capability of the model. For the precision metric and F-1 score the confidence level is on their optimal confidence thresholds.

Table 4.2: Baseline vs. HPO-tuned model performance.

Model	**Precision**	**Recall**	**mAP@0.5**	**F1**
Baseline	0.82	0.44	0.564	0.51
HPO-tuned	0.91	0.51	0.623	0.58

As shown in Table 4.2, the HPO tuned model outperforms the baseline across all metrics. The most notable gains are observed in precision (+0.09), recall (+0.07), and F1 score (+0.07), indicating that the tuned model not only makes more correct detections but also misses fewer objects. The last one is particularly important in the target detection scenario, where missed detections carry a higher cost than false positives. The mAP@0.5 also improved from 0.564 to 0.623 (+0.059), reflecting a better overall precision-recall trade-off across all confidence thresholds. To determine whether these improvements are statistically meaningful rather than a result of random variation, a significance test is conducted in the following section.

Figure 4.1: Example of HPO tuned model inference on UAV imagery.

Figure 4.1 demonstrates an example of the HPO tuned model inference on UAV imagery. In (a), the model successfully detects multiple instances of `civilian_person` in close proximity, with confidence scores reaching up to 0.88. In (b), the model identifies several `civilian_vehicle` instances in a top-down aerial view, with confidence scores ranging from 0.69 to 0.77. These examples illustrate the model's ability to handle both pedestrian and vehicle detection across different aerial perspectives.

4.3 Statistical Significance

To validate that the observed performance gains are not due to random variation, a Wilcoxon signed-rank test was applied on per-class AP@0.5 values across 10 classes. The test yielded a p-value of 0.003843, which is well below the significance threshold of α=0.05, confirming that the improvement is statistically significant.

The effect size, measured by Cohen's d, was 0.693, which corresponds to a large effect. The mean per-class improvement in AP@0.5 was +0.1144, with a 95% bootstrap confidence interval of [+0.0302, +0.2245], meaning the true improvement is real and consistent across classes.

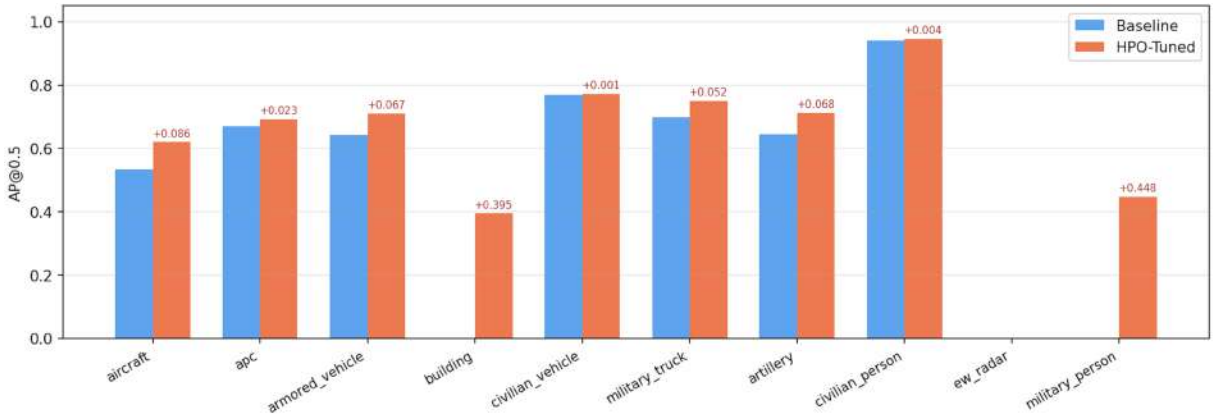


Figure 4.2: Per-class AP@0.5 (validation set).

As illustrated in Figure 4.2, the HPO tuned model shows consistent gains across most classes. The largest improvements were observed for `building` (+0.395), `military_person` (+0.448), and `aircraft` (+0.086), while `civilian_vehicle` and `ew_radar` showed minimal change. Overall, these results confirm that hyperparameter optimization produced a meaningful and statistically significant improvement to the model's detection performance.

Further work and Conclusions

Conclusions

This work investigated the application of automated hyperparameter optimization to improve object detection performance on UAV aerial imagery. A two-stage HPO pipeline was constructed, combining random search for broad exploration with the Tree-structured Parzen Estimator for focused refinement, totalling 20 optimization trials. The pipeline was applied to a YOLOv8n model trained on a custom aerial dataset of 20,209 images across 10 object classes.

The optimized model demonstrated consistent improvement over the baseline across all evaluated metrics, with mAP@0.5 increasing from 0.564 to 0.623. Statistical significance testing confirmed that these gains are meaningful and not a result of random variation, with a Wilcoxon p-value of 0.003843 and a large effect size (Cohen's d = 0.693).

While the results are promising, the current system is not yet at a production-ready level. Further improvements, outlined in the following section, are needed before deployment in real-world scenarios.

Further Work

The most impactful direction would be expanding the training dataset. The current dataset, while sufficient for initial experimentation, lacks coverage of objects from diverse viewing angles and distances. This is likely a contributing factor to missed detections, particularly for underrepresented classes such as `ew_radar`. As shown in Table 3.1, the `ew_radar` class contain only 39 annotations, compared to a dataset average of 1,846 annotations per class. This severe class imbalance is directly reflected in Figure 4.2, where `ew_radar` achieves an AP@0.5 of 0.0 for both the baseline and HPO-tuned models. In contrast, well-represented classes such as `civilian_person` (1,142 annotations) achieve AP@0.5 of 0.93, suggesting that annotation volume is a primary bottleneck for underrepresented classes. Collecting more data that captures objects from a wider range of UAV altitudes and trajectories would directly improve the model's generalization ability.

A closely related direction is data augmentation. Since collecting real aerial footage is costly, augmentation techniques specifically designed for aerial imagery — such as random perspective transforms, altitude simulation, and mosaic augmentation — could synthetically expand the dataset and expose the model to a greater variety of viewpoints.

Finally, data annotation quality and consistency could be further improved. Expanding the set of annotated frames and ensuring uniform labeling standards across all classes would reduce noise in the training signal and potentially improve performance on minority classes.

All these directions collectively would bring the model closer to production-level performance for real-world UAV deployment.

Use of AI Tools

AI-assisted tools were used during several stages of this work. Claude Sonnet 4.6 (Anthropic) was used for writing and debugging code, as well as drafting and refining thesis text. Perplexity AI was used for searching relevant academic sources. Google Gemini was used for formatting content in LaTeX. All analytical conclusions, experimental design, and final decisions were made independently by the author.

Bibliography

[1] Z. Zou et al. “Object Detection in 20 Years: A Survey”. In: *arXiv* (2023).

[2] I. Hossain. *Support Vector Machine*. Tech. rep. Frankfurt University of Applied Sciences, 2022. URL: https://www.researchgate.net/publication/365892847_Support_Vector_Machine.

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

[4] G. Tang et al. “A Survey of Object Detection for UAVs Based on Deep Learning”. In: *Remote Sensing* 15.15 (2023), p. 3733. DOI: 10.3390/rs15153733.

[5] Annotation Jlzgv. *Detection Dataset*. Roboflow Universe. Accessed: 2026-05-13. 2026. URL: https://universe.roboflow.com/annotation-jlzgv/detection-ysdlh/browse.

[6] Kırıkkale Üniversitesi. *SIHA Dataset*. Roboflow Universe. Accessed: 2026-05-13. 2026. URL: https://universe.roboflow.com/krkkale-niversitesi-pnmrg/siha-easlk/dataset/3.

[7] P. Liashchynskyi and P. Liashchynskyi. “Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS”. In: *arXiv* (2019). URL: https://arxiv.org/abs/1912.06059.

[8] J. Bergstra and Y. Bengio. “Random Search for Hyper-Parameter Optimization”. In: *Journal of Machine Learning Research* 13.1 (2012), pp. 281–305. URL: https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.

[9] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in Neural Information Processing Systems (NIPS)*. Vol. 24. 2011, pp. 2546–2554. URL: https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619db63395a-Abstract.html.

[10] A. Ng. *Gaussian Processes*. Tech. rep. Stanford University, 2018. URL: https://see.stanford.edu/materials/aimlcs229/cs229-gp.pdf.

[11] Ultralytics. *Hyperparameter Tuning with Ultralytics YOLO*. Ultralytics Documentation. Accessed: 2026-05-13. 2026. URL: https://docs.ultralytics.com/guides/hyperparameter-tuning/.

[12] M. Yaseen. "What is YOLOv8: An In-depth Exploration of the Internal Features of the Next-generation Object Detector". In: *arXiv* (2024).

[13] D. Reis, J. Kupec, J. Hong, and A. Daoudi. "Real-Time Flying Object Detection with YOLOv8". In: *arXiv* (2023). URL: https://arxiv.org/abs/2304.00501.

[14] IBM. *What is a Loss Function?* IBM Think. Accessed: 2026-05-13. 2024. URL: https://www.ibm.com/think/topics/loss-function.

[15] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[16] X. Li et al. "Generalized Focal Loss: Learning Qualified and Distributed Bounding Boxes for Dense Object Detection". In: *arXiv* (2020).

[17] S. Watanabe et al. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *arXiv* (2023).

[18] Evidently AI. *Accuracy, Precision, and Recall in Machine Learning*. Evidently AI. Accessed: 2026-05-13. 2023. URL: https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall.

[19] T.-Y. Lin et al. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision (ECCV)*. 2014, pp. 740–755.