

# AIR

Airbus AI Research

---

Université Paul Valéry, Montpellier III UFR 6



Master's Degree in Mathematics and Computer Science Applied to Human and Social Sciences

Trustworthy AI for Autonomous Vision-Based Landing : Assessing Robustness, Explainability, and Conformal Prediction for Runway Detection.

Supervised by Mélanie Ducoffe and Jérôme Pasquet

2024-2025

by Alya ZOUZOU

# Table des matières

<b>List of figures</b>	<b>2</b>
<b>Introduction</b>	<b>5</b>
<b>Acknowledgements</b>	<b>5</b>
<b>Lexicon</b>	<b>6</b>
<b>1 The Industrial Context</b>	<b>8</b>
1.1 From the Current Landing . . . . .	8
1.2 ... to vision-based landing . . . . .	9
1.2.1 The general objective of the project . . . . .	9
1.2.2 Specifications of the Vision Landing System . . . . .	10
1.2.3 Why not simply use conventional solutions ? . . . . .	11
<b>2 Presentation of the data : LARD</b>	<b>12</b>
2.1 Previous approaches and their shortcomings . . . . .	12
2.2 How the Dataset Works . . . . .	13
2.2.1 A generic definition of a landing . . . . .	13
2.2.2 Data diversity . . . . .	14
<b>3 State-Of-The-Art in Object Detection : YOLO</b>	<b>17</b>
3.1 Why YOLO ? . . . . .	18
3.2 Architecture . . . . .	18
3.2.1 The Backbone . . . . .	18
3.2.2 Neck & Head . . . . .	20
3.3 How does a detection work ? . . . . .	20
3.4 Standard evaluation in object detection . . . . .	21
3.4.1 mAP : Mean Average Precision . . . . .	21
3.4.2 Definitions of IoU and IoA . . . . .	22
3.5 How is a YOLO dataset formatted ? . . . . .	23
3.5.1 TXT files (Annotations) . . . . .	23
3.5.2 The YAML file (Configuration) . . . . .	24
3.6 How to read a YOLO tensor ? . . . . .	24
3.7 Creating the YOLO dataset . . . . .	25
3.7.1 Architecture of the LARD Dataset . . . . .	25
3.7.2 Creating labels and YOLO dataset structure . . . . .	25

<b>4 Training and Performance Analysis</b>	<b>27</b>
4.1 Training Runs . . . . .	27
4.1.1 Overview of the tools and approach . . . . .	27
4.1.2 Optuna : Hyperparameter optimization . . . . .	28
4.1.3 Running the trainings . . . . .	28
4.1.4 Currently fine-tuned model bank . . . . .	30
<b>5 Robust vision-based runway detection using conformal prediction and a conformal mAP.</b>	<b>32</b>
5.1 Objective and Contributions . . . . .	32
5.2 What is conformal prediction ? . . . . .	33
5.2.1 Conformal prediction for object detection . . . . .	34
5.3 How to evaluate a robust detection ? . . . . .	37
5.3.1 Are IoU and IoA robust metrics ? . . . . .	37
5.3.2 C-mAP : Conformal Mean Average Precision . . . . .	38
5.3.3 Other evaluation metrics . . . . .	39
5.4 Experiments . . . . .	40
5.4.1 Data and model training . . . . .	40
5.4.2 Model configurations and training . . . . .	40
5.4.3 Results . . . . .	40
5.5 Conclusion . . . . .	42
<b>6 POSEIDON : POSe Estimation with ExplIcit Differentiable OptimizatioN</b>	<b>43</b>
6.1 Context and Objectives . . . . .	43
6.2 Clarification on the Term <i>Pose</i> . . . . .	44
6.3 Pose-Oriented Detection with YOLO-NAS-POSE . . . . .	45
6.3.1 Architecture Overview . . . . .	45
6.3.2 Neural Architecture Search (NAS) . . . . .	48
6.3.3 Quantization-Aware Training (QAT) . . . . .	48
6.3.4 Data Format and Preparation for YOLO-NAS-POSE . . . . .	49
6.3.5 Losses . . . . .	50
6.3.6 Fine-Tuning YOLO-NAS-POSE on LARD . . . . .	51
6.4 Toward an End-to-End Pipeline for Pose Estimation . . . . .	53
<b>Conclusion</b>	<b>56</b>
<b>Annexes</b>	<b>57</b>
<b>7 Appendix</b>	<b>58</b>
.1 Training Details and Example Predictions for the Different Configurations . . . . .	58
.1.1 How to launch a training run ? . . . . .	58
.1.2 Example of Metadata . . . . .	59
.1.3 Example Predictions for the Configurations . . . . .	60
.2 C-mAP Implementation in Python . . . . .	61
.3 Details on How the YOLO-NAS-POSE Losses Work . . . . .	62
.4 Detailed Explanation of P3P following Kneip et al. [2011] . . . . .	64

# Table des figures

1.1	A Landing.	8
1.2	A Runway.	8
1.3	Global Landing System using GPS.	9
1.4	Instrument Landing System.	9
1.5	Pipeline of VLS	10
2.1	Geometry of a Landing.	13
2.3	Runway with different distances.	14
2.4	3-dimensional visualisation of aircraft positions in the training set and the test set.	14
2.5	Illustration of the quality of the synthetic images - Comparison of a real landing footage (left) with a synthetic replica (right)	15
2.6	Proportion of each subset of the dataset	16
3.1	The BackBone.	18
3.2	BottleNeckCSP Block	19
3.3	SiLU and Sigmoid Activation function	19
3.4	Interpolated Average Precision curve.	22
3.5	Illustration of IoU and IoA metrics between a predicted box and a true box.	23
3.6	Step-by-step computation of mAP on an example from the LARD dataset. The image at the top shows predictions at different confidence thresholds, while the table details the changes in precision and recall.	23
3.7	Label Examples.	26
4.1	Dataset size on the AWS bucket.	27
4.2	Adversarial Example Ilyas et al. [2019]	29
4.3	Labelled Batch as Reference.	30
4.4	Maximum mAP's of each model through the trials.	30
4.5	mAP's evolution of each model through the trials.	31
5.1	Illustration of conformal prediction applied to runway detection. Red : ground truth, blue : YOLO prediction, green : enlarged conformal box.	33
5.2	Illustration of expanding the predicted box into a conformal box.	36
5.3	Conformal boxes (in green) obtained with the additive method for YOLOv5 (left) and YOLOv6 (right).	36
5.4	Conformal boxes (in green) obtained with the multiplicative method for YOLOv5 (left) and YOLOv6 (right).	36

5.5	Illustration of the <i>coverage</i> metric : the ground truth (in red) is fully included in the conformal box (in orange). . . . .	37
5.6	Comparison of two cases : left, the ground truth is covered ( $\text{IoA} = 1$ ) but poorly localized (low IoU) ; right, both metrics are satisfied. . . . .	38
5.7	Computation of <b>C-(m)AP</b> step-by-step on an example from the LARD dataset. The top image shows the visual setup, while the table illustrates how predictions at different confidence levels affect precision and recall. . . . .	39
6.1	Illustration of the pose estimation problem from Kneip et al. [2011] (P3P) . . . . .	45
6.2	Example prediction on an image of dancers . . . . .	45
6.3	YOLO-NAS Backbone architecture . . . . .	46
6.4	YOLO-NAS-POSE extension . . . . .	47
6.5	Linear mapping of FP32 values to INT8 via symmetric quantization. . . . .	48
6.6	Labeling . . . . .	50
6.7	Qualitative comparison of YOLO-NAS-POSE-S outputs on LARD. . . . .	52
1	Prediction Example for V5 Nano Pretrained during Trial 1. . . . .	60
2	Prediction Example for V5 Small Pretrained during Trial 1. . . . .	60
3	Prediction Example for V5 Nano Random during Trial 2. . . . .	60
4	Prediction Example for V5 Small Random during Trial 2. . . . .	61
5	Illustration of pose estimation from Kneip et al. [2011] (P3P). . . . .	64
6	Our feature vectors. . . . .	66
7	Image basis associated with the direction vectors. . . . .	67

# Introduction

Airbus is a renowned aerospace manufacturer founded in the 1970s. It is a major European company known for its innovations and market share in the field of air transport. Indeed, Airbus produces nearly half of the world's civil airliners. In 2023, the company delivered 735 commercial aircraft, achieved a turnover of 65.4 billion euros, and a profit of 5.8 billion euros. The most significant branch, Airbus Commercial Aircraft, specializes in civil airliners, including medium-haul aircraft such as the A320, the best-selling airliner in the world.

In my second year, I am part of the Artificial Intelligence Research Team 1XRD, led by Jayant Sen Gupta, within Airbus's Central Research and Technology (CRT). This team is dedicated to research and innovation in the fields of aeronautics and defense, in collaboration with internationally renowned schools, universities, and research centers. My supervisor, Mélanie Ducoffe, is a senior researcher specializing in AI, particularly in robustness, certification, and explainability in the field of computer vision applied to critical systems. She works on approaches aimed at ensuring that AI systems are safe, reliable, and certifiable, which are major challenges in aeronautics and other sectors requiring resilience in their systems.

In this context, I will contribute to the work of the CRT and the DEEL team (Dependable, Certifiable Explainable AI for Critical Systems), a research program of the ANITI laboratory dedicated to the development of robust and certifiable artificial intelligence technologies, specifically designed for critical systems. My project focuses on the study and application of object detection and computer vision technologies in the context of autonomous landing.

Indeed, Airbus plans to develop single-pilot cockpits by 2035. This development primarily aims to support the pilot in critical tasks and enhance the autonomy of the onboard systems.

Working within the 1XRD team at Airbus, in collaboration with DEEL, is a unique opportunity to contribute to research projects in the field of aeronautics while learning and deepening my knowledge.

# Acknowledgments

I would like to express my deep gratitude to my mentor, Mélanie Ducoffe, for her constant support, valuable advice, and kindness throughout this work. Her expertise and encouragement have been of great help and have significantly contributed to the progress of this project.

I also thank my professor, Jérôme Pasquet, for his involvement, constructive feedback, and availability.

Finally, I wish to thank the entire teaching staff of the MIASHS program, whom I have had the chance to learn from since my first year of undergraduate studies up to my Master's degree. Thank you for your teachings, encouragement, and support throughout these years.

# Lexicon

- **CRT** : Center of Research and Technology
- **DEEL** : Dependable, Certifiable & Explainable AI for Critical Systems
- **YOLO** : You Only Look Once (algorithme de détection d'objets)
- **VLS** : Visual Landing System
- **GPS** : Global Positioning System
- **LARD** : Landing Approach Runway Detection : Dataset for Vision-Based Landing
- **EASA** : European Union Aviation Safety Agency
- **FAA** : Federal Aviation Administration
- **SPPF** : Spatial Pyramid Pooling - Fast
- **SOTA** : State of the Art. It refers to the best performance or the most advanced method known to date in a given field.
- **ODD** : Operational Design Domain
- **CP** : Conformal Prediction
- **IoA** : Intersection Over Area
- **IoU** : Intersection Over Union
- **NAS** : Neural Architecture Search
- **NMS** : Non-Maximum Suppression
- **End-to-End** : approach where a system learns directly to perform a complete task, from raw input data to the final output, without manual intermediate steps.
- **PNP** : Perspective with N points
- **P3P** : Perspective with 3 points
- **GPU** : Graphics processing unit

# Chapitre 1

## The Industrial Context

### Sommaire

<b>1.1 From the Current Landing...</b>	<b>8</b>
<b>1.2 ... to vision-based landing.</b>	<b>9</b>
1.2.1 The general objective of the project	9
1.2.2 Specifications of the Vision Landing System	10
1.2.3 Why not simply use conventional solutions ?	11

### 1.1 From the Current Landing...

Landing 1.1 is a critical phase of aviation, requiring perfect coordination between pilots, onboard systems, and ground infrastructures. Currently, commercial aircraft operate with two pilots, who work in cooperation with control towers and ground systems. This redundancy is essential to ensure safety in case of human or technical failure. In summary, landing is currently based on :

- **Two pilots** : One is responsible for the main maneuvers (pilot in command), while the other supervises the systems and performs a secondary check.
- **Runway Vocabulary** : Communications with the control tower rely on a standardized language (ATIS) to avoid misunderstandings, particularly during landing.
- **Infrastructures** : Runways are equipped with markings, light beacons, and systems such as ILS (Instrument Landing System) and GPS (Global Positioning System).



FIGURE 1.1 – A Landing.



FIGURE 1.2 – A Runway.

## 1.2 ... to vision-based landing.

### 1.2.1 The general objective of the project

Safety remains Airbus's priority. In a few years, the company aims to ensure that, even if pilots are no longer able to control the aircraft, autonomous systems based on AI can ensure a safe landing. The first step is to consider a single-pilot cockpit and to introduce vision-based landing technology as **a pilot assistance only**. Before being implemented on board, this technology must obtain certification in accordance with EASA (European Agency ????) standards. Certification imposes constraints on embarkability, robustness, and explainability. But also the adherence to two fundamental principles in aviation : redundancy and resilience.

**Redundancy** Redundancy involves duplicating critical systems to ensure their operation even in the event of a failure. For landing, two redundant systems, in addition to the pilots, are currently used :

- **GPS** : The GPS uses a network of satellites to provide a global location of the aircraft 1.3. However, its accuracy decreases significantly near the runway, with an error margin of about 30 meters, which limits its use in the final phases of landing. Furthermore, the increased GPS signal disruptions due to military conflicts in certain regions underscore the need to develop mitigation means to ensure greater reliability.

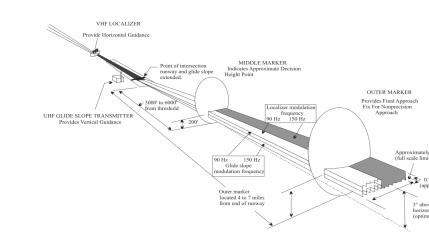
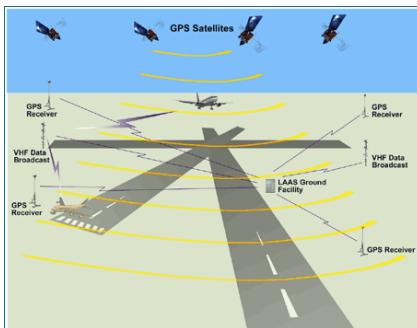


FIGURE 1.4 – Instrument Landing System.

FIGURE 1.3 – Global Landing System using GPS.

- **ILS** : The ILS system relies on ground antennas that emit radio signals providing precise lateral guidance (runway centerline) and vertical guidance (glide path) 1.4. Although several precision levels exist (CAT-1, CAT-2, CAT-3), the infrastructure required for the highest levels, such as CAT-3, remains costly to install and maintain, which limits implementation across all airports. Moreover, although modern aircraft are equipped with features enabling fully automated landings, their use has a significant impact on airport operations, notably by reducing the number of possible landings per hour, which poses a challenge for overall operational efficiency.

**Resilience** refers to a system's ability to maintain or restore its functioning in the face of disruptions or failures. It is based on two fundamental principles :

- **Independent redundancy** : Integrating multiple systems whose information modes and failure modes are independent, in order to reduce the risk of simultaneous failures affecting all systems.
- **Functional recovery** : The ability to restore normal operation after an incident, which requires the presence of at least three redundant systems. This configuration allows a majority or median logic to be applied to ensure reliable decision-making.

To ensure resilience and redundancy in the event of pilot incapacitation, a vision-based system, VLS, appears to be an appropriate solution to complement ILS and GPS. However, its deployment entails several major industrial challenges :

- **Operational performance** : Ensuring that the system meets performance requirements, notably its ability to estimate the aircraft's position in real time during landing, so as to guarantee a precise and reliable convergence to the runway, even under complex conditions.
- **Certification and compliance** : Being able to certify such a tool in accordance with current standards governing the use of artificial intelligence in industrial products. This step is essential to guarantee the system's reliability, safety, and regulatory acceptance.

### 1.2.2 Specifications of the Vision Landing System

The VLS pipeline breaks down into three main stages, each playing a role in the aircraft detection and localization process :

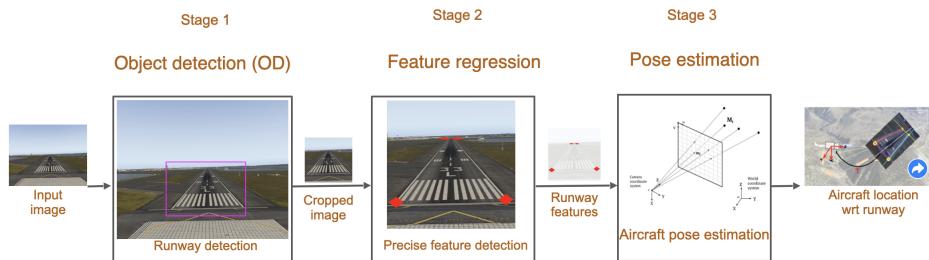
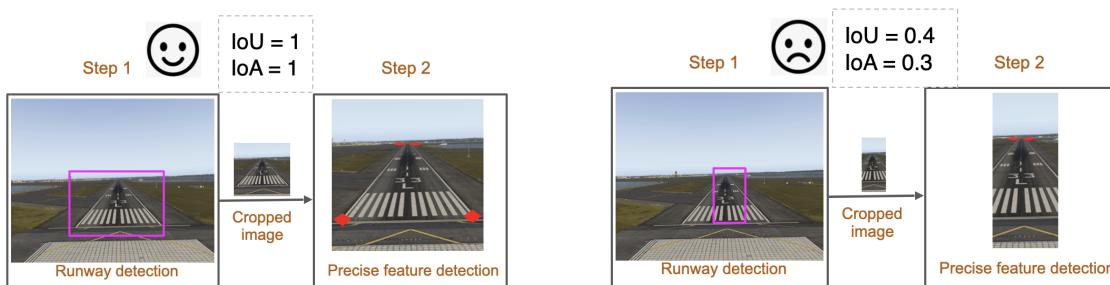


FIGURE 1.5 – Pipeline of VLS

- **Stage 1 — Object Detection (OD)** : The input 1.5 is an image in which the system detects the runway, as illustrated in the first part of the pipeline. The image is cropped around the area of interest to isolate the runway more precisely. This stage therefore requires training object detection algorithms that perform well on the specific task of runway detection.
- **Stage 2 — Feature Regression** : Once the image has been cropped 1.5, precise runway features, such as the four corners, are extracted. This helps characterize the runway geometry and prepare the data for the next stage.
- **Stage 3 — Pose Estimation** : This stage determines the aircraft's exact position relative to the runway using a 3D coordinate system and a pose-estimation procedure, enabling the aircraft's location to be known in real time during landing.

These three stages are crucial for ensuring accurate detection and reliable estimation of the aircraft's position, even under challenging environmental conditions. The first task—runway detection—is particularly important because it correctly identifies the landing area before moving on to the next step, extracting precise features from the cropped image. Indeed, poor initial detection will yield an image that does not contain the information needed for feature extraction, thereby degrading the accuracy of the aircraft position estimate.



That said, although these steps may seem “trivial” at first glance, the main challenge lies in certifying this technology with aviation authorities. For instance, EASA (European Union Aviation Safety Agency) imposes numerous constraints to ensure safety in European aviation. Before entering service, aircraft must be certified in accordance with the requirements of various international organizations, such as the Federal Aviation Administration (FAA) or the Civil Aviation Authority of China (CAAC).

Given the complexity and breadth of these constraints, it is impossible for me to list them all here, especially since they often involve interconnected systems requiring aeronautical expertise that I do not possess. I will therefore limit myself to the main requirements related to my task :

- **Computational complexity and accuracy** : The system must operate with minimal latency, enabling detection at **20 frames per second**, i.e., nearly in real time. This constraint is particularly demanding due to the absence of integrated GPUs in safety-critical systems, which limits the execution of compute-intensive calculations at high precision. Although specialized chips designed to combine low power consumption with intensive computation are under development, their use in this field remains confidential and falls under industrial secrecy, preventing me from providing further details.
- **Embeddability** : The models must be compatible with the ONNX format to ensure integration into resource-constrained embedded systems. This format plays a key role in optimizing models for constrained environments. It is nevertheless expected that these formats will evolve to adapt to new chips under development, enabling more efficient inference while respecting the strict constraints of safety-critical systems.
- **Robustness** : The system must be capable of accurately identifying three-dimensional landmarks—specifically longitude, latitude, and Perspective-n-Point (PnP) parameters—even under difficult conditions such as nighttime, low visibility, or environments affected by extreme weather (rain, fog). It must also remain effective in the face of technical issues such as camera pixel dropouts or sensor errors. This robustness is essential to maintain reliable navigation during the critical phases of landing.
- **Explainability** : Beyond ensuring redundancy of hardware and software systems, it is essential to guarantee the explainability of models deployed in safety-critical environments. This means that decisions made by the algorithms—particularly those related to autonomous navigation—must be understandable and justifiable, just as a pilot must explain their choices during a landing. Explainability makes it possible to verify system reliability in the event of an incident and to identify the reasons behind an algorithmic decision, which is crucial for validation, certification, and building trust in these systems. “

### 1.2.3 Why not simply use conventional solutions ?

Using GPS coordinates or QR codes may at first seem like an intuitive solution. However, several constraints make these approaches inapplicable :

- **Resolution issues** : At a distance of 6 nautical miles (about 3 km), the runway occupies only 20 pixels in a standard image. This makes detection based on simple GPS coordinates imprecise.
- **Infrastructure limitations** : Since Airbus has no control over airports, it is impossible to mandate the installation of QR codes or other specific markings on runways.

## Chapitre 2

# Presentation of the data : LARD

### Sommaire

---

2.1	Previous approaches and their shortcomings . . . . .	12
2.2	How the Dataset Works . . . . .	13
2.2.1	A generic definition of a landing . . . . .	13
2.2.2	Data diversity . . . . .	14

---

### 2.1 Previous approaches and their shortcomings

Earlier work on runway detection mainly used methods based on image processing or machine-learning algorithms such as SVMs and AdaBoost. These approaches focused on identifying runway-specific features, such as geometric patterns or textures, often leveraging classical transforms such as Sobel or Canny filters.

However, these methods have several major limitations :

- **Lack of robustness** : These methods are sensitive to disturbances, notably the presence of objects near the runway.
- **Limited generalization** : The algorithms struggle to adapt to varied environments and to detect runways under difficult conditions such as fog, night, or rain. It is important to note that in the event of a failure in visual detection, other redundancy systems — such as GPS, ILS (Instrument Landing System), or assistance from air-traffic controllers — support the pilots to ensure a safe landing.
- **Lack of reproducibility** : Many studies rely on private databases, which limits their applicability and the comparison of results.

These limitations motivated the creation of a representative Open-Source dataset, **LARD** Ducoffe et al. [2023] (Landing Approach Runway Detection), whose objective is clear : *detect a runway in an image taken during the landing phase of an aircraft*.

The rest of my explanations are based on the **LARD** paper Ducoffe et al. [2023], with a personal reformulation of the concepts. It seems crucial to me to understand how this database was constructed, as this not only clarifies the scale of the challenge, but also places runway detection as a *primary* step in the overall autonomous-landing process. The latter indeed requires a combination of multiple interconnected technologies to ensure the system's safety, reliability, and full autonomy.

## 2.2 How the Dataset Works

### 2.2.1 A generic definition of a landing

Landing corresponds to the final phase in which an **aircraft** approaches and converges toward a **runway** to touch down. To analyze this maneuver effectively, it must be represented in a standardized way so as to preserve all characteristic information in the image. This phase relates two main elements : the aircraft and the runway, whose positions change over time. The LARD paper proposes a generic description of these interactions to enable their modeling and analysis in vision-based systems.

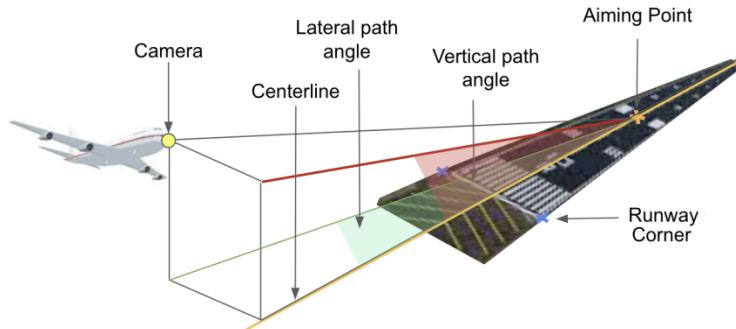


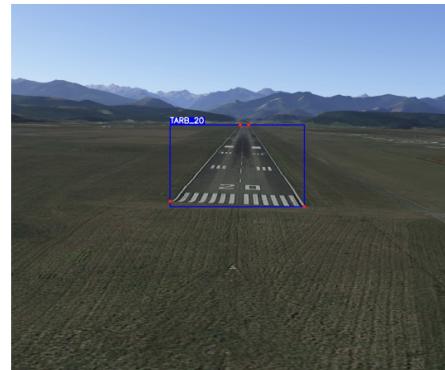
FIGURE 2.1 – Geometry of a Landing.

#### The runway

The runway is defined by its four corners, which makes it possible to draw a *bounding box* in the images.



(a) A Runway with plane on it.



(b) What the OD Model should detect.

The runway corners are determined from data such as GPS coordinates, then projected into the images via geometric transformations. Here we describe what a runway looks like and how its different zones are used. The blast-protection area is generally located before the start of the runway. It also serves as an emergency space in the event of problems during takeoff, providing an extra stopping extension if necessary. This area is marked by yellow chevrons. Runway markings are standardized and often include an initial line called the *runway threshold*, which marks the beginning of the takeoff or landing zone. This line is generally followed by striped patterns called *the piano keys or runway threshold* as well as identifiers visible on both sides of the runway centerline.

On runways longer than 1500 m, a set of wide rectangular bars indicates the target point called the “aiming point” where the aircraft will touch down during landing.

The main operational distances are :

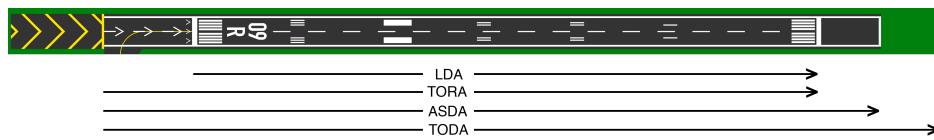


FIGURE 2.3 – Runway with different distances.

- **LDA (Landing Distance Available)** : portion of runway usable for landing, from the threshold to the end of the stop zone.
- **TORA (Takeoff Run Available)** : length available for takeoff acceleration, which may include a displaced threshold.
- **ASDA (Accelerate-Stop Distance Available)** : distance to accelerate and then stop the aircraft in the event of a rejected takeoff, including the stopway.
- **TODA (Takeoff Distance Available)** : total distance available for takeoff, combining acceleration and climb-out, including extensions and clearways.

#### The approach cone

The approach cone is based on six main parameters : the distance to the runway (*along track distance*), the lateral and vertical path angles (*lateral* and *vertical path angles*), as well as the aircraft attitude (pitch, roll, yaw). This ensures a realistic representation of landing scenarios and facilitates data annotation.

In summary, the landing trajectory targets the *Aiming Point*, located 300 meters beyond the runway threshold, between two rectangular markings. The aircraft's position relative to the runway is defined by three parameters :

- **Distance along the runway (*along track distance*)** : Distance between the projection of the aircraft's nose onto the runway centerline and the *Aiming Point*.
- **Lateral and vertical angles** : Formed respectively between the runway axis and the line connecting the *Aiming Point* to the aircraft's nose, in ground-plan or vertical-plane projection.
- **Aircraft attitude** : Defined by the pitch (longitudinal inclination), roll (lateral inclination), and yaw (orientation with respect to the runway axis).

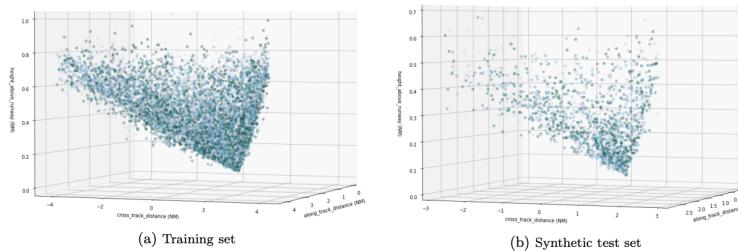


FIGURE 2.4 – 3-dimensional visualisation of aircraft positions in the training set and the test set.

#### 2.2.2 Data diversity

- The **LARD** dataset is designed to represent a wide range of landing scenarios thanks to two main data sources :
- **Synthetic images** : Produced mainly via Google Earth Studio, these images offer high resolution and automatic annotations. It makes it possible to produce satellite-style images enriched with metadata (runway position, camera position and angle, etc.) from configuration files. The latter define the capture parameters (time, weather, angles, etc.) and can be used to automatically generate an image sequence using Google Earth Studio.

Category	Parameters
<b>Camera</b>	Position (Longitude, Latitude, Altitude) Rotation (Horizontal angle, Vertical angle, Roll) Field of view
<b>Environment</b>	Date (Year, Month, Day, Hour)
<b>Rendering</b>	Output type (Images .jpeg or video .mp4) Dimensions (Width, Height) Metadata (3D positions in .json format) Image quality (Texture)
<b>Attribution</b>	Attribution credit position

TABLE 2.1 – Configurable parameters in Google Earth Studio

The process includes generating a scenario file, using it in Earth Studio to produce the images, then automatically annotating the images using geometric projections. This annotation relies on the extrinsic matrices (camera position and orientation) and intrinsic matrices (optical parameters), making it possible to project the coordinates of the runway corners into the image frame. This pipeline makes it possible to generate a very large number of annotated images from a single initial annotation, which drastically reduces labeling cost.

- **Real images annotated manually :** From cockpit landing videos, they enhance the dataset with precise annotations of runway corners, including cases where the aircraft is misaligned with the runway. This diversity strengthens representativeness of real scenarios, notably in situations where the aircraft is not perfectly parallel to the runway.



FIGURE 2.5 – Illustration of the quality of the synthetic images - Comparison of a real landing footage (left) with a synthetic replica (right)

The two types of data allow models to be trained on synthetic images while testing their ability to generalize to real data (*Sim-to-Real*).

### Some figures

The dataset consists of the following sets :

- **Training set :** Exclusively synthetic, it comprises **14 433 images** with a resolution of  $2448 \times 2648$ , captured over 32 runways across 16 airports, i.e., about 451 images per approach. A subset of **5 runways** is dedicated to domain adaptation (*train\_da*) which can be used as a validation set if necessary.
- **Test set :** It includes **2 315 images**, of which :
  - **Synthetic images :** **2 221 images**, from 79 runways in 40 different airports, representing about 28 images per approach.
  - **Real images :** **103 annotated images** from 38 runways in 36 airports, captured using onboard cameras. These images are divided into nominal cases, edge cases, and domain-adaptation subsets.

The synthetic test images include varied environments and airport types not encountered during training. The real images make it possible to evaluate the models' generalization and adaptation capabilities to real-world data.

A balanced distribution of runway centers is observed across all the data, except for a sparsely represented area at the top and bottom of the images, linked to cropping margins.



FIGURE 2.6 – Proportion of each subset of the dataset

## Chapitre 3

# State-Of-The-Art in Object Detection : YOLO

### Sommaire

---

<b>3.1</b>	<b>Why YOLO ?</b>	<b>18</b>
<b>3.2</b>	<b>Architecture</b>	<b>18</b>
3.2.1	The Backbone	18
3.2.2	Neck & Head	20
<b>3.3</b>	<b>How does a detection work ?</b>	<b>20</b>
<b>3.4</b>	<b>Standard evaluation in object detection</b>	<b>21</b>
3.4.1	mAP : Mean Average Precision.	21
3.4.2	Definitions of IoU and IoA	22
<b>3.5</b>	<b>How is a YOLO dataset formatted ?</b>	<b>23</b>
3.5.1	TXT files (Annotations)	23
3.5.2	The YAML file (Configuration)	24
<b>3.6</b>	<b>How to read a YOLO tensor ?</b>	<b>24</b>
<b>3.7</b>	<b>Creating the YOLO dataset</b>	<b>25</b>
3.7.1	Architecture of the LARD Dataset	25
3.7.2	Creating labels and YOLO dataset structure	25

---

### 3.1 Why YOLO ?

YOLO (*You Only Look Once*) Redmon et al. [2016] is a family of object detection models developed by **Ultralytics**. The introduction of YOLOv5 in 2020 marked a major advance thanks to its migration to PyTorch, bringing better flexibility and ease of use. YOLOv5 stands out for its efficiency, speed, and ability to operate in real time. It is therefore our choice to evaluate performance and establish a benchmark for the **LARD** dataset. I will detail here how YOLOv5 works, which is the model I used most during this first semester. That said, many concepts apply to more recent YOLO architectures. To establish the benchmark, I will need to evaluate the performance of several other architectures (YOLO v6 Li et al. [2023], v8 Varghese and M. [2024]...).

YOLOv5 comes in several model sizes to meet different accuracy and resource needs :

Model	Description	Number of parameters
<b>YOLOv5n</b>	the lightest, ideal for embedded systems.	~2,5 MB
<b>YOLOv5s</b>	balance between speed and accuracy	7,2M
<b>YOLOv5m</b>	Good trade-off between accuracy and speed.	21,2M
<b>YOLOv5l</b>	Designed for tasks requiring higher accuracy.	46,5M
<b>YOLOv5x</b>	The most accurate, optimized for small-object detection.	86,7M

TABLE 3.1 – The different YOLOv5 model sizes

As a reminder, in constrained environments such as aeronautics, large models may be unsuitable due to onboard constraints. That is why we will only use the Nano and Small models.

### 3.2 Architecture

The YOLOv5 architecture Khanam and Hussain [2024] is built on three main components : the *Backbone*, the *Neck*, and the *Heads*. These three components extract visual features, fuse them, and predict the objects present in an image.

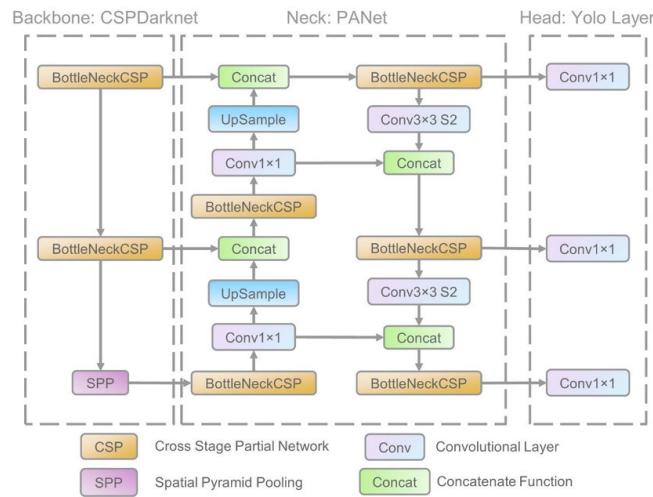


FIGURE 3.1 – The BackBone.

#### 3.2.1 The Backbone

The YOLOv5 Jocher [2020] *Backbone* is based on the **CSPDarknet** (*Cross Stage Partial Darknet*) structure, designed to extract features from images while minimizing computational redundancy. The main Backbone compo-

nents are :

- **BottleNeckCSP** : This block combines residual and partial connections to improve information flow and reduce redundant gradients. These connections make it possible to efficiently train deep networks while avoiding *vanishing gradients*. Within CSPNet, the feature map is split into two distinct parts in the “partial dense block” (3.2, right). The first part passes through dense layers, where it is duplicated : one part undergoes convolutions to extract new features, while the other is preserved as is, without modification. These two outputs are then concatenated to enrich the extracted features. The base (unmodified) feature map ( $x_0$ ) is directly concatenated at the transition layer to preserve the base information. This mechanism improves feature reuse while avoiding excessive gradient duplication, which optimizes computation and maintains high model accuracy.

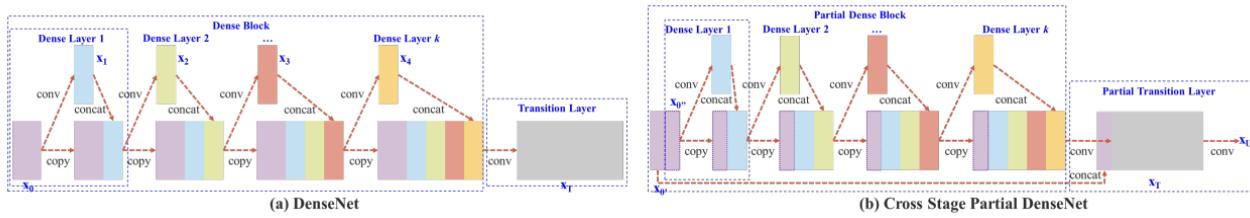


FIGURE 3.2 – BottleNeckCSP Block

- **SiLU activation function** : The **SiLU** (Sigmoid Linear Unit) function is used to improve the network’s learning capacity, with the formula :

$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}.$$

The SiLU activation preserves negative values while attenuating them, making it a more flexible alternative to ReLU, which zeros out all negative outputs. It is used in the hidden layers of the network *backbone*. At the network output, another activation function is used : the sigmoid. It is applied to objectness scores and class probabilities, which must be between 0 and 1. For bounding-box coordinates, specific transformations (often sigmoid-based or offsets) are used to constrain predictions into a logical space.

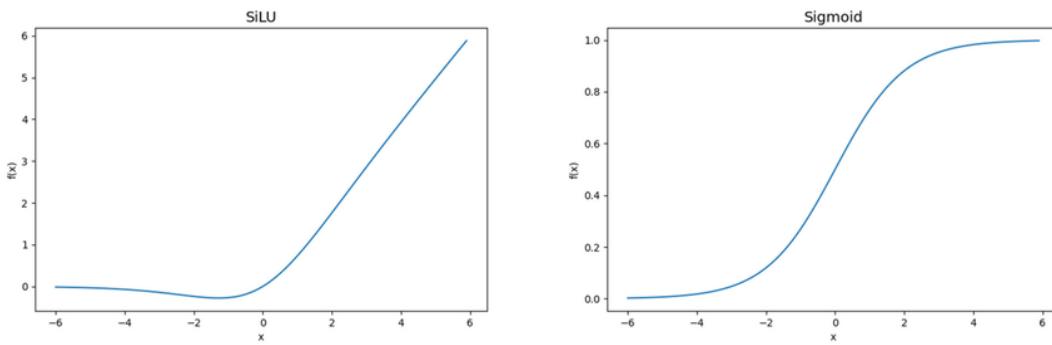


FIGURE 3.3 – SiLU and Sigmoid Activation function

- **Batch Normalization (BN)** : This technique stabilizes intermediate activations and speeds up learning by regularizing the outputs of convolutional layers. It operates in two steps : a normalization of activations using the mini-batch mean and variance,

$$\hat{x}_i = \frac{x_i - \mu_{\text{batch}}}{\sqrt{\sigma_{\text{batch}}^2 + \epsilon}},$$

followed by an affine transformation,

$$y_i = \gamma \hat{x}_i + \beta,$$

where  $\gamma$  and  $\beta$  are parameters learned during training.

### 3.2.2 Neck & Head

The *Neck* connects the *Backbone*, which extracts image features, to the *Heads*, which make predictions about objects present at different resolution levels. The *Backbone* captures features at different image scales as network depth increases, and the *Neck* fuses these features to capture objects of varying sizes.

The *Neck* consists mainly of two elements : **PANet** and **SPPF**.

- **PANet (Path Aggregation Network)** : PANet fuses features extracted at different resolutions to better detect small objects. It improves the way information flows between the model's different layers, enabling it to better capture fine details that may be important for detecting smaller objects.
- **SPPF (Spatial Pyramid Pooling - Fast)** : SPPF increases the model's receptive field without slowing it down. This means the model can take into account a larger portion of the image at once, enabling it to better understand relationships between objects, even across scales.

In summary, the *Neck* in YOLOv5 is crucial to efficiently combine information extracted at different resolutions, improving detection of objects of all sizes. PANet and SPPF work together to ensure the model is both fast and accurate, even in complex scenes with variably-sized objects.

**Head** : The *Heads* are responsible for the final prediction of bounding boxes, confidence scores, and associated classes.

In YOLOv5, the *Head* produces outputs at multiple resolutions to enable the detection of objects at different sizes. Thus, the output tensor is structured according to three resolution levels corresponding to different spatial scales from the *Neck* :

- A **high-resolution** output (e.g.,  $80 \times 80$  for a  $640 \times 640$  input image) to detect small objects.
- An **intermediate-resolution** output (e.g.,  $40 \times 40$ ) to detect medium-sized objects.
- A **low-resolution** output (e.g.,  $20 \times 20$ ) to detect large objects.

## 3.3 How does a detection work ?

When a detection model analyzes an image, it typically generates a large number of candidate boxes, some of which may overlap heavily. To quantify this overlap, we use the *Intersection over Union* (IoU) metric, defined as :

$$\text{IoU} = \frac{|B_{\text{predicted}} \cap B_{\text{true}}|}{|B_{\text{predicted}} \cup B_{\text{true}}|}$$

To remove redundancy, a step called **non-maximum suppression** (NMS Neubeck and Van Gool [2006]) is applied. It keeps only the most confident boxes when they overlap beyond a defined threshold. A second step then filters out boxes considered unreliable based on their objectness score.

We formalize the detection model as a function  $f$ , composed of two elements : the detector with NMS and a confidence threshold. The detector with NMS is defined as follows :

$$f^{\text{NMS}} : x \rightarrow \left( \hat{\mathbf{b}}_i, o_i, \mathbf{p}_i \right)_{i=1}^{N_x^{\text{NMS}}}$$

where :

- $\hat{\mathbf{b}}_i$  is the predicted bounding box for the  $i$ -th detection,
- $o_i$  represents the objectness score (probability of an object being present),
- $\mathbf{p}_i$  is a vector indicating the probabilities for each class.

Only detections with an objectness score greater than a threshold  $\tau$  are kept. We then define :

$$I(x) = \{i \in \{1, \dots, N_x^{\text{NMS}}\} : o_{[i]} \geq \tau\}$$

where  $o_{[i]}$  corresponds to scores sorted in descending order. In other words,  $I(x)$  contains the indices of predictions whose confidence is high enough to be retained.

The final model output is then :

$$f : x \rightarrow \{f^{\text{NMS}}(x)_{[k]}, k \in I(x)\}$$

This process retains only predictions deemed sufficiently reliable while eliminating duplicates and unlikely false positives.

## 3.4 Standard evaluation in object detection

### 3.4.1 mAP : Mean Average Precision.

The evaluation of object detection models is primarily based on the **mean Average Precision (mAP)** metric, which combines both the evaluation of classification quality and the accuracy of object localization. This measure relies on the spatial similarity between predicted boxes and ground-truth boxes, in terms of *Intersection over Union (IoU)*.

Consider a set of predicted boxes  $\{\hat{\mathbf{b}}_i^x\}$  produced by the detector for an image  $x$ , and a set of ground-truth annotations  $\{(\mathbf{b}_j^x, c_j^x)\}_j$ , where  $\mathbf{b}_j^x$  represents a true annotated box with class  $c_j^x$ . The boxes  $\hat{\mathbf{b}}_i^x$  correspond to the final predictions retained *post NMS* (non-maximum suppression), i.e., the most likely and non-redundant boxes for the image in question.

Let  $C$  be the total number of classes ; mAP is defined as :

$$\text{mAP} = \frac{1}{C} \sum_{c=1}^C AP \left( \{\hat{\mathbf{b}}_i^x\}_{\hat{c}_i^x=c}, \{\mathbf{b}_j^x\}_{c_j^x=c} \right), \quad \text{where } \hat{c}_i^x = \arg \max_k \mathbf{p}_i^x(k)$$

In our case, the mAP evaluation concerns a single class—the runway—which amounts to focusing only on **Average Precision (AP)**, with  $C = 1$ .

AP computation is based on matching predicted boxes with ground-truth annotations using an IoU threshold. A prediction is considered a **true positive (TP)** if its IoU with a true box exceeds a certain threshold  $\tau$ , typically set to 0,5. Unmatched predicted boxes are counted as **false positives (FP)**, while undetected true boxes constitute **false negatives (FN)**.

Predictions are then sorted by decreasing confidence score and used to build an **interpolated precision-recall curve**. Recall the definitions :

— **Precision :**

$$p = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

— **Recall :**

$$r = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The precision-recall (PR) curve plots precision as a function of recall. In object detection, the problem is discrete—with a finite number  $n$  of predictions—so this curve is constructed from  $n$  points, corresponding to different ranking thresholds of the predictions. To facilitate the calculation of the area under the curve, and thus AP, an interpolation is generally applied to smooth the curve. The classical interpolation takes, for each recall level  $r$ , the maximum precision achieved for any recall greater than or equal to  $r$  :

$$\bar{p}(r) = \max_{\tilde{r} \geq r} p(\tilde{r})$$

**Average Precision (AP)** is then defined as the area under this interpolated curve. A historical approximation method is the *11-point interpolation*, which evaluates the interpolated precision at recall levels  $r \in \{0.0, 0.1, \dots, 1.0\}$  :

$$AP = \frac{1}{11} \sum_{r \in \{0.0, 0.1, \dots, 1.0\}} \bar{p}(r)$$

In modern benchmarks, such as COCO, finer interpolations are used. One often reports the **mAP@50 :95** metric, which is the mean of AP computed for IoU thresholds from 0,50 to 0,95, with a step of 0,05.

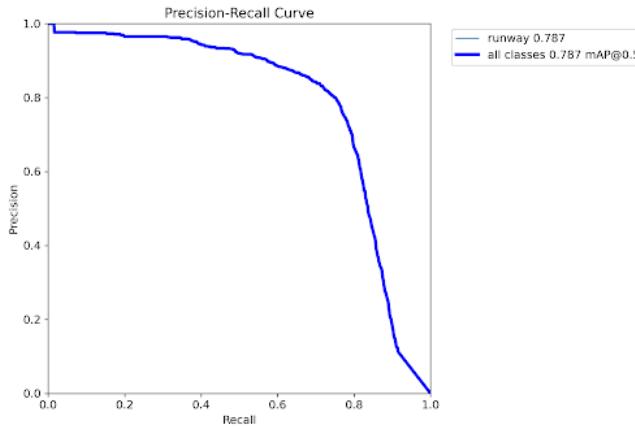


FIGURE 3.4 – Interpolated Average Precision curve.

### 3.4.2 Definitions of IoU and IoA.

*Intersection over Union (IoU)* is a standard overlap measure between two bounding boxes : a predicted box  $B_p$  and a ground-truth box  $B_{gt}$ . It is defined as the ratio between the area of the intersection of the two boxes and the area of their union :

$$\text{IoU}(B_p, B_{gt}) = \frac{\text{area}(B_p \cap B_{gt})}{\text{area}(B_p \cup B_{gt})}$$

This metric evaluates the spatial localization accuracy of a prediction. An IoU threshold (generally 0,5) is then set to determine whether a prediction is considered a **true positive (TP)**.

We also sometimes distinguish another measure : *Intersection over Area (IoA)*, which compares the intersection to the area of the true box only. It is defined as :

$$\text{IoA}(b_{\text{pred}}, b_{\text{gt}}) = \frac{\text{area}(b_{\text{pred}} \cap b_{\text{gt}})}{\text{area}(b_{\text{gt}})}$$

This measure can be useful when we want to guarantee a minimum coverage of the ground truth, regardless of the prediction size.

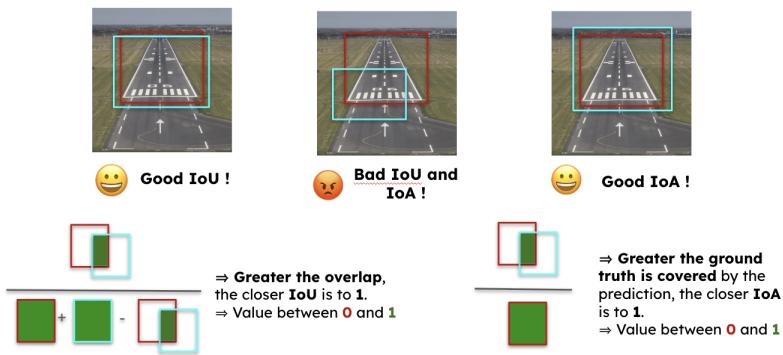


FIGURE 3.5 – Illustration of IoU and IoA metrics between a predicted box and a true box.

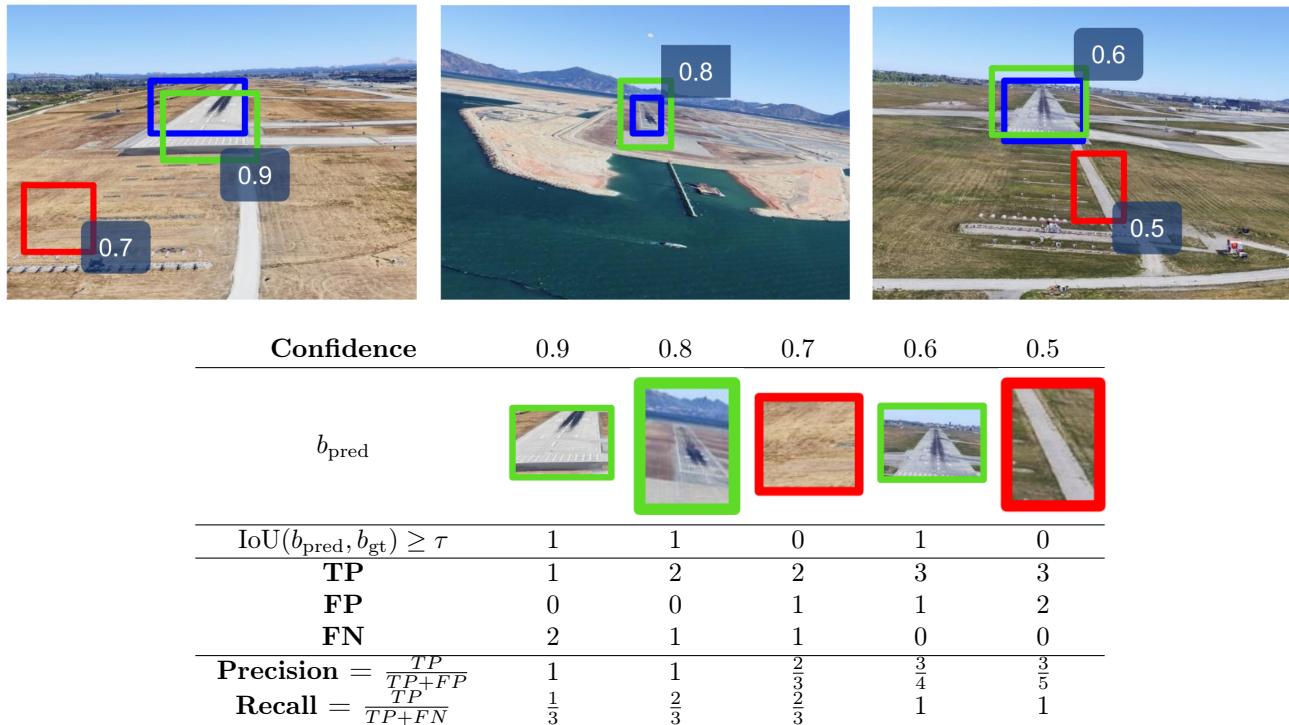


FIGURE 3.6 – Step-by-step computation of mAP on an example from the LARD dataset. The image at the top shows predictions at different confidence thresholds, while the table details the changes in precision and recall.

### 3.5 How is a YOLO dataset formatted ?

A YOLO-style dataset relies on two types of files to structure the data : **-txt** files containing annotations and a **yaml** file for the dataset's global configuration.

#### 3.5.1 TXT files (Annotations)

Each line of a TXT file corresponds to an annotated object and contains :

1. Class ID (an integer between 0 and (Number of classes - 1)) : e.g., 0 for “person”.
2.  $x$ -coordinate of the box center (between 0 and 1).
3.  $y$ -coordinate of the box center (between 0 and 1).

4. Width of the bounding box (between 0 and 1).
5. Height of the bounding box (between 0 and 1).

Here is an example. If an image contains a person and a car, the TXT file associated with that image could look like this :

$$\begin{bmatrix} 0 & 0.486 & 0.652 & 0.3 & 0.6 \end{bmatrix} \quad \begin{bmatrix} 1 & 0.322 & 0.435 & 0.25 & 0.35 \end{bmatrix}$$

Here 0 indicates the presence of a person in the image and 1 the presence of a car.

### 3.5.2 The YAML file (Configuration)

The YAML file contains the project's global information :

```

1 path: /path/to/dataset
2 train: /path/to/train/images # Path to training images.
3 test: /path/to/train/images # Path to test images.
4 val: /path/to/val/images # Path to validation images.
5 nc: 2 # Number of classes (here, 2)
6 names: ['person', 'car'] # associates a name with the class id

```

## 3.6 How to read a YOLO tensor ?

After the image passes through the network, a prediction *tensor* is produced. The following information is provided for each detected object :

- **Bounding-box coordinates** : Each box is defined by its center ( $x, y$ ), width  $w$ , and height  $h$ , often normalized with respect to the image dimensions.
- **Objectness score** : Probability that an object is present in the predicted box (value between 0 and 1).
- **Classes** : Probabilities for each class, indicating which category the detected object belongs to.
- **Confidence score** : Product of objectness and class probability, used to filter less reliable predictions.

For example, in an image where two objects are detected : a person (class 0) and a tie (class 27). The output tensor could look like this :

$$\begin{bmatrix} 0.5 & 0.5 & 0.3 & 0.4 & 0.85 & 0 & 0.9 \end{bmatrix} \quad \begin{bmatrix} 0.8 & 0.6 & 0.2 & 0.3 & 0.75 & 27 & 0.8 \end{bmatrix}$$

Each row corresponds to a prediction :

- **First row** : A person (class 0) with an objectness score of 0.85 and a class probability of 0.9.
- **Second row** : A tie (class 27) with an objectness score of 0.75 and a class probability of 0.8.

Each prediction in the tensor follows this structure :

1. **xmin** : Minimum  $x$  coordinate (top-left corner).
2. **ymin** : Minimum  $y$  coordinate (top-left corner).
3. **xmax** : Maximum  $x$  coordinate (bottom-right corner).
4. **ymax** : Maximum  $y$  coordinate (bottom-right corner).
5. **Objectness score** : Probability that an object is present in this box.
6. **Classes** : List of probabilities for each object class.

**Global confidence** : The overall confidence score is not explicitly present in the tensor, but can be computed by multiplying objectness by the probability of the dominant class. This score is used to apply a confidence threshold and filter detections.

## 3.7 Creating the YOLO dataset

### 3.7.1 Architecture of the LARD Dataset

The LARD dataset contains about 17,000 images, organized into main folders for training and testing :

**Training folders :** The LARD\_train\_\* folders contain image subfolders and a csv file with the following columns :

- **image, height, width** : Basic image information, including relative path, height, and width.
- **type** : Data type, either **real** (real) or **earth\_studio** (synthetic).
- **original\_dataset, scenario, airport, runway** : Information about the image source, such as airport and targeted runway.
- **weather, night** : Indicate reduced visibility (rain) and whether the image was captured at night.
- **time\_to\_landing** : Remaining time (in seconds) before reaching the target point on the runway (real images only).
- **slant\_distance, along\_track\_distance, height\_above\_runway, lateral\_path\_angle, vertical\_path\_angle** : Metadata specific to synthetic images, describing the aircraft's position relative to the runway.
- **Runway corner coordinates** : **x\_A, y\_A, x\_B, y\_B, x\_C, y\_C, x\_D, y\_D**.

All synthetic images have a resolution of 2448×2648, while real images mainly vary between 3840×2160 and 1920×1080.

**Test files :**

- **LARD\_test\_real/** : Contains subfolders for **nominal\_cases**, **edge\_case** (low visibility) and **domain\_adaptation**.
- **LARD\_test\_synth/** : Dedicated to testing models on synthetic images.

**Metadata file :** Each folder contains a CSV file describing the associated metadata, and a global file **LARD\_train.csv** aggregates all training-image information. An example of metadata can be found in Appendix .1.2.

#### Details : Real vs synthetic images

Real images provide information about time to landing, weather conditions, and time of day (**night**), with runway corner coordinates extracted directly. Synthetic images simulate night by reducing brightness, but do not contain weather data.

### 3.7.2 Creating labels and YOLO dataset structure

Next, I used a Python script to generate YOLO-format annotations for each image in the LARD dataset. The goal was to create TXT annotation files for each image, indicating the presence of the runway. Since the only class to detect is *runway*, the class identifier (ID) is always set to 0. In addition, we are sure that a detected runway is unique in the image.

First, I configured local paths to save annotations in the **train** and **val** folders. Then, the script accesses the CSV files stored on an S3 bucket using **boto3**, and for each row it extracts the necessary information, such as runway corner coordinates and image dimensions.

The bounding-box coordinates are then normalized for the YOLO format. The **normalize\_coordinates** function computes the center and size of the box relative to the image size. Here is an excerpt from this function :

```

1 def normalize_coordinates(x_A, y_A, x_B, y_B, x_C, y_C, x_D, y_D, image_width, image_height):
2     x_min = min(x_A, x_B, x_C, x_D)
3     x_max = max(x_A, x_B, x_C, x_D)
4     y_min = min(y_A, y_B, y_C, y_D)
5     y_max = max(y_A, y_B, y_C, y_D)
6
7     # Compute normalized center and size
8     x_center = (x_min + x_max) / 2 / image_width
9     y_center = (y_min + y_max) / 2 / image_height

```

```
10     width = (x_max - x_min) / image_width  
11     height = (y_max - y_min) / image_height  
12     return x_center, y_center, width, height
```

Then, for each image, I generated a YOLO-format annotation text file with class ID 0 and the normalized coordinates of the box center and size. The resulting file was saved in the correct folder (`train` or `val`) depending on the name of the CSV file.

The YOLO format for each annotation is therefore :

$$[0 \ x\_center \ y\_center \ width \ height]$$

where 0 represents the *runway* class ID, and the other values are normalized coordinates.

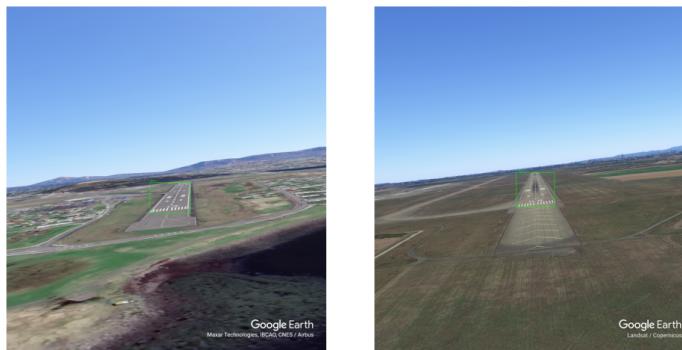


FIGURE 3.7 – Label Examples.

## Chapitre 4

# Training and Performance Analysis

## Sommaire

<b>4.1 Training Runs . . . . .</b>	<b>27</b>
4.1.1 Overview of the tools and approach . . . . .	27
4.1.2 Optuna : Hyperparameter optimization . . . . .	28
4.1.3 Running the trainings . . . . .	28
4.1.4 Currently fine-tuned model bank . . . . .	30

## 4.1 Training Runs

### 4.1.1 Overview of the tools and approach

After converting the LARD database to the YOLO format, I first studied the YOLOv5 algorithm to understand its workings and specificities. I then set up a complete Python environment, using Ultralytics' `requirements.txt` and installing all the necessary libraries.

Mastering the CUDA libraries was essential to leverage GPU acceleration—indispensable for training on a dataset as large as ours : 33,498 objects for 49.8 GB (see Figure 4.1). I therefore set up an AWS cloud environment equipped with a GPU and stored the dataset on an S3 bucket.

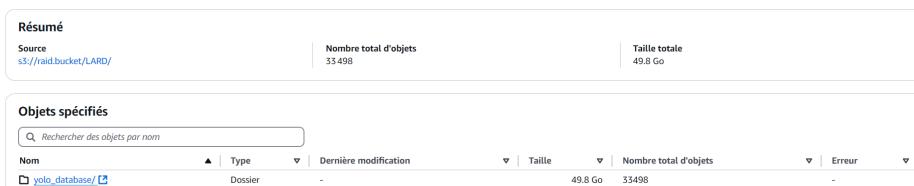


FIGURE 4.1 – Dataset size on the AWS bucket.

Once the environment was ready, I launched four « naïve » training runs with manually chosen hyperparameters to check that everything worked correctly and to obtain a baseline. However, this method does not achieve the best performance, since data quality must be complemented by fine hyperparameter tuning.

That is why I decided to integrate automatic hyperparameter optimization into my training, in order to identify the best-performing configurations for the architectures used. All trainings were carried out under similar hardware conditions to ensure a fair and reliable comparison.

### 4.1.2 Optuna : Hyperparameter optimization

Optuna Akiba et al. [2019] is an open-source library dedicated to hyperparameter optimization, a crucial step for maximizing model performance (accuracy, training speed, etc.). Deep neural networks are particularly sensitive to these parameters (learning rate, number and size of layers, optimizer type, etc.), and poor settings can lead to slow convergence or mediocre performance.

Unlike grid or random search, Optuna uses Bayesian optimization that intelligently selects subsequent configurations based on previous results, exploring the hyperparameter space more efficiently. This approach reduces compute time and focuses on promising regions.

We chose Optuna for its ease of use, clear documentation, and straightforward integration into our Python environment. Optimization targeted the **mAP0.5** metric as the primary objective.

### 4.1.3 Running the trainings

Full details of the trainings performed with Ultralytics are provided in Appendix .1.1.

#### Training with pre-trained weights vs random initialization

We chose to train our models either with pre-trained weights or with random initialization. In machine-learning model training, using pre-trained weights leverages models that have already learned representations on large, often general-purpose datasets. However, this approach can create a dependency on *features* that are not directly correlated with the target task. These features, while highly predictive, can be fragile and not interpretable by humans, which introduces safety risks—especially in critical domains such as aeronautics.

One underlying problem with using these pre-trained weights 4.2 is that the model may learn biased—and thus adversarial—features : patterns in the data that are predictive but not robust. Such characteristics can be specific to contexts irrelevant to the task, as in computer-vision models that learn traits related to objects like cats, which have no value in an aeronautical context. The problem here is that these non-robust features are likely to be manipulated or vulnerable to adversarial attacks : imperceptible changes that can induce large prediction errors. In aeronautics, this represents a major risk, because a small change in the runway image or environment could disrupt the system’s ability to recognize the situation correctly—unacceptable in a safety-critical context. Ilyas et al. Ilyas et al. [2019] show that these non-robust features are potentially the source of adversarial examples—often unintelligible to humans but exploitable by the model—which constitutes a safety risk.

Another approach is random initialization of model weights 4.2, which avoids introducing irrelevant features from pre-existing datasets. Although this method can lead to slower convergence and require more data, it ensures that the model is not biased by adversarial characteristics. This can strengthen robustness to unforeseen perturbations, which is crucial in sensitive applications such as aeronautics.

In conclusion, the choice between using pre-trained weights and random initialization directly concerns the reliability and safety of AI systems in critical domains. The decision should therefore balance model performance with guarantees of robustness against adversarial risks.

#### How to launch a training run with Optuna ?

In this section, I describe the methodology used for fine-tuning models with pre-trained weights and for training from randomly initialized weights. The approaches differ depending on the configurations.

#### Fine-tuning the *nano* and *small* models with pre-trained weights

This approach aims to leverage pre-trained weights to optimize the performance of the *YOLO v5 nano* and *small* models, while identifying the best hyperparameters with Optuna.

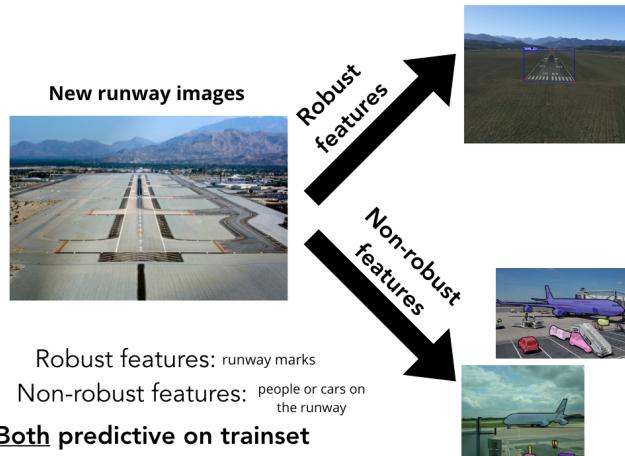


FIGURE 4.2 – Adversarial Example Ilyas et al. [2019]

The optimized hyperparameters include : `lr` (learning rate), `momentum`, `weight_decay` (regularization), `batch` (batch size), `epochs` (number of epochs) and `imgsz` (image size). For each trial, Optuna generates a combination of hyperparameters that is passed to a Python script (`yolo_train_v5.py`) via command-line arguments. Evaluation is based on mAP50 (*mean Average Precision at IoU=0.5*), computed at each trial. This metric is saved in a `pkl` file and used to guide optimization.

To avoid GPU memory saturation errors when using pre-trained weights together with Optuna optimization, the procedure was split across two separate scripts :

- `yolo_train_v5.py`: This script handles training manually, taking hyperparameters as command-line arguments. It returns mAP50, used as the optimization objective.
- `optim_v5_nano_pretrained.py`: This script orchestrates Optuna optimization. Each trial generates a set of hyperparameters that is passed to the training script and waits for the mAP to be returned. Passing these arguments via the command line simplifies their retrieval and use during optimization.

#### Fine-tuning the *nano* and *small* models with random weight initialization

This section details the training process for *YOLO v5 nano* and *small* models with randomly initialized weights. The main objective remains the same : optimize hyperparameters to maximize performance in terms of *mean Average Precision at IoU=0.5* (mAP@0.5) via Optuna.

Training begins by dynamically creating the hyperparameters for each trial. These parameters also include the learning rate (`lr`), momentum (`momentum`), and weight decay (`weight_decay`). A dedicated function generates a YAML configuration file (`hyp.yaml`), which is then used by the training script within a single file. In the case of randomly initialized weights, the GPU did not saturate easily, so I was able to use the Ultralytics file that launches training and perform the optimization in a specific file.

The YOLOv5 `train.py` script is executed on the command line, receiving the generated hyperparameters as arguments. To ensure random initialization, empty weights (`weights=''`) are specified, and a suitable configuration (`yolov5s.yaml` for the *small* model) is used.

Training metrics, including mAP@0.5, are recorded in a CSV file generated by the training script. A specific function extracts the mAP@0.5 value from the last line of this file, allowing model performance to be evaluated for each trial. This value is then used as the return value in Optuna's objective function to guide the optimization.

Optuna explores different hyperparameter combinations, automatically adjusting trials to maximize mAP@0.5. This identifies the most effective parameters for models trained from randomly initialized weights.

The files and scripts used in this process include :

- `train.py`, the Ultralytics file that manages model training with the provided hyperparameters.

- `hyp.yaml`, a configuration file generated dynamically for each trial before optimization.
- `optim_v5_nano_random.py`: a script that launches training via CLI using the `-hyp` argument and passes mAP directly to Optuna's objective function within the same file.

#### 4.1.4 Currently fine-tuned model bank

In this section, I focused exclusively on evaluating YOLOv5 and its four variants : *nano random*, *nano pretrained*, *small random*, and *small pretrained*. The goal is to analyze the performance of each configuration by highlighting their predictions on a representative data sample, while quantifying results using the mAP@0.5 metric.

For each model, an example prediction *batch* is presented, allowing us to visualize effectiveness on the tested data. These visualizations illustrate qualitative differences between the trained configurations and initial weights, whether random or pre-trained.

Figure 4.3 below shows an example of a labeled *batch*, which serves as the reference baseline for the predictions produced by the studied models.



FIGURE 4.3 – Labelled Batch as Reference.

Finally, you can find example predictions for each studied configuration in Appendix .1.3

#### Model Comparison

In the first chart 4.4, we show the maximum mAP reached for each model :

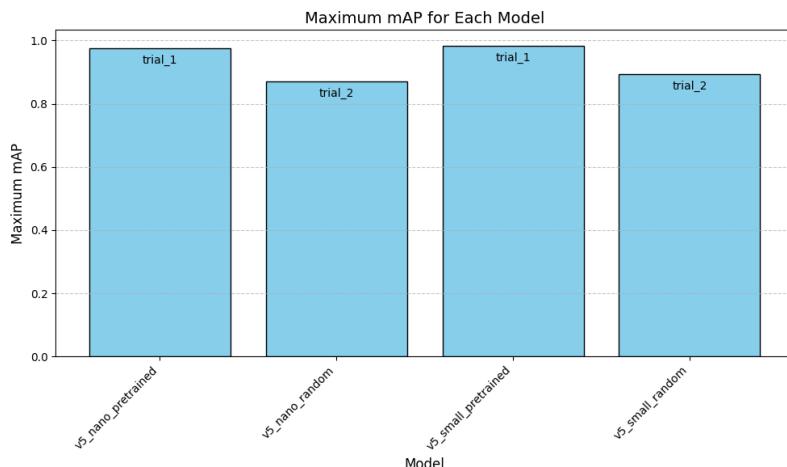


FIGURE 4.4 – Maximum mAP's of each model through the trials.

- `v5_small_pretrained` delivers the best performance, reaching an mAP close to 1.0.
- `v5_nano_pretrained` follows with similar but slightly lower performance.
- The randomly initialized models (`v5_nano_random` and `v5_small_random`) show significantly lower mAP, indicating difficulty converging as effectively as the pre-trained ones.

This trend confirms that using pre-trained weights provides a considerable advantage, likely due to better parameter initialization that accelerates and stabilizes learning. It is important to note that this gap can be reduced with more extensive training, since we limited the epoch optimization range from 10 to 50.

The second chart 4.5 shows the evolution of mAP for each model over several trials (0 to 9) :

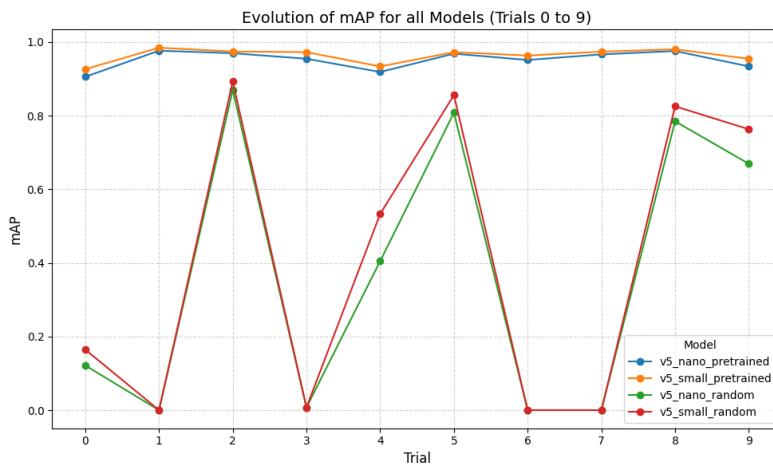


FIGURE 4.5 – mAP's evolution of each model through the trials.

- The pre-trained models (`v5_nano_pretrained` and `v5_small_pretrained`) display rapid and stable progress from the earliest trials. They quickly converge to high mAP values, often close to 1.0.
- The random models (`v5_nano_random` and `v5_small_random`) show unstable and unpredictable performance. The mAP fluctuates strongly, with periods where they fail to learn properly, as evidenced by drastic drops (e.g., trials 2, 4, and 7).

The instability of random models suggests that learning is more sensitive to hyperparameters or data variability.

In conclusion, these results highlight the importance of pre-trained weights for runway detection tasks. Pre-trained models deliver more stable and higher performance in less time. Conversely, randomly initialized models require more effort in hyperparameter tuning and longer training to hope to rival the pre-trained models.

## Chapitre 5

# Robust vision-based runway detection using conformal prediction and a conformal mAP.

### Sommaire

---

<b>5.1</b>	<b>Objective and Contributions . . . . .</b>	<b>32</b>
<b>5.2</b>	<b>What is conformal prediction ? . . . . .</b>	<b>33</b>
5.2.1	Conformal prediction for object detection . . . . .	34
<b>5.3</b>	<b>How to evaluate a robust detection ? . . . . .</b>	<b>37</b>
5.3.1	Are IoU and IoA robust metrics ? . . . . .	37
5.3.2	C-mAP : Conformal Mean Average Precision . . . . .	38
5.3.3	Other evaluation metrics . . . . .	39
<b>5.4</b>	<b>Experiments . . . . .</b>	<b>40</b>
5.4.1	Data and model training . . . . .	40
5.4.2	Model configurations and training . . . . .	40
5.4.3	Results . . . . .	40
<b>5.5</b>	<b>Conclusion . . . . .</b>	<b>42</b>

---

### 5.1 Objective and Contributions

In Chapter 1, I presented runway detection as a crucial step in the VLS pipeline (see Fig. 1.5). Indeed, the image is cropped according to the *bounding box* produced by this detection, forming the only source of information for the following stages :

- extracting the four corners of the runway,
- estimating the aircraft’s position.

Detection accuracy is therefore essential : if the crop does not contain at least three runway corners, position estimation becomes impossible, compromising the system’s operation.

This led me to ask : *how can we quantify the risk and uncertainty associated with this detection ?*

To answer this, I explored the use of **conformal prediction** to provide statistical guarantees on the uncertainty associated with runway detection. By applying this method to **YOLOv5** and **YOLOv6** models fine-tuned on the **LARD** dataset, I evaluated the reliability of detections for a defined risk level. I also proposed a new metric, **Conformal Mean Average Precision (C-mAP)**, which combines detection performance with conformal guarantees.

The results show that conformal prediction significantly improves the reliability of detections by rigorously quantifying the system's statistical uncertainty.

This work<sup>1</sup> led to the paper "*Robust Vision-Based Runway Detection through Conformal Prediction and Conformal mAP*"<sup>2</sup>, accepted for an oral presentation at the COPA (Conformal and Probabilistic Predictions with Applications) conference in London in September 2025.

This work aimed to strengthen the robustness of baseline detectors such as YOLO—high-performing on standard tasks but less precise in critical situations—and to quantify uncertainty by integrating robustness into classical evaluation metrics such as IoU (Intersection over Union) and mAP (mean Average Precision).



FIGURE 5.1 – Illustration of conformal prediction applied to runway detection. Red : ground truth, blue : YOLO prediction, green : enlarged conformal box.

The main goal is to define a conformal predictor that envelops the ground truth with risk control, via a controlled expansion of the box predicted by the base model using conformal prediction.

## 5.2 What is conformal prediction ?

In machine learning, predictive models, built from data, remain subject to uncertainty that can affect their predictions.

We distinguish two types of uncertainty :

- **Aleatoric uncertainty** : irreducible, linked to the inherent variability of the phenomenon (weather, lighting, unpredictable conditions). In runway detection, it reflects natural scene variations, impossible to eliminate even with a better model.
- **Epistemic uncertainty** : reducible, due to a lack of data or poor diversity. The LARD set, mostly synthetic with few night views and a single runway, limits generalization. A more diverse dataset would reduce this uncertainty.

Quantifying this uncertainty—whatever its origin—is therefore crucial, especially in applications where safety and performance are essential. Conformal prediction is a relatively recent approach that estimates this uncertainty and provides more robust and reliable predictions.

**Conformal Prediction (CP)** is thus a method that estimates uncertainty by constructing valid prediction sets. A prediction set is said to be valid if it guarantees marginal probabilistic coverage. In other words, a prediction set contains the true value of a test point with a pre-defined probability, regardless of the model used.

Its main advantages are :

- **Distribution-free** : no assumption on the data distribution is needed, ensuring validity even without knowledge of the underlying model.

1. [https://github.com/alyasltd/conformal\\_runway\\_detection](https://github.com/alyasltd/conformal_runway_detection)  
 2. <https://arxiv.org/abs/2505.16740>

- **Model-agnostic** : applicable to any algorithm, including “black-box” models, without needing to know their internal architecture.
- **Small-sample guarantees** : provides rigorous probabilistic guarantees even with a limited number of observations, unlike asymptotic approaches.

However, for these guarantees to hold, some minimal assumptions must be met :

- **Exchangeability assumption** : data must be drawn from a time-stable distribution, independently and identically distributed (i.i.d.), or at least be exchangeable.
- **Independence between training and calibration sets** : data used to train the model must be independent from the data used to calibrate predictions.
- **Distributional homogeneity** : test data must follow the same distribution as the calibration set.

Thus, let  $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$  be a training set of exchangeable points  $(x_i, y_i)$ , where  $x_i$  denotes features and  $y_i$  the corresponding labels. We assume these data are generated from the same joint distribution.

Conformal prediction proceeds as follows :

1. A predictive model is first trained on the training data.
2. For each test point  $x_{\text{new}}$ , a set of possible values  $\hat{C}_\alpha(x_{\text{new}})$  is constructed. This set is such that the true value  $y_{\text{new}}$  lies in it with probability at least  $1 - \alpha$ , where  $\alpha$  is the tolerated risk level.

Mathematically, for an error rate  $\alpha$ , the probability that the prediction set for a test point contains the ground truth is at least  $1 - \alpha$ .

$$P(\{Y_{\text{new}} \in \hat{C}_\alpha(X_{\text{new}})\}) \geq 1 - \alpha$$

where  $\hat{C}_\alpha(X_{\text{new}})$  is the prediction set for  $X_{\text{new}}$ , and  $\alpha$  is the user-chosen risk level. This framework guarantees that, across multiple calibration and test sets, the prediction set will contain the true value with frequency at least  $1 - \alpha$ .

### 5.2.1 Conformal prediction for object detection

Applied to object detection, **conformal prediction** associates a statistically guaranteed confidence with boxes generated by models like YOLO, even in a “black-box” setting. This is particularly relevant for the LARD dataset, where the limited number of test examples makes the method’s small-sample guarantees valuable.

However, certain assumptions must hold : exchangeability is acceptable for image-by-image analysis, but not for a temporally correlated video stream as in a VLS, where CP is then limited to an evaluative role. Moreover, calibration and test sets must share the same distribution, which is ensured here via homogeneous sampling from LARD.

Thus, despite constraints specific to object detection, CP remains applicable and enables more robust evaluation by attaching explicit confidence to each prediction.

#### Mathematical formalization

Conformal prediction can be adapted to object detection tasks, with the objective of providing bounding boxes endowed with probabilistic guarantees. Several formulations exist for this extension. In our case, we adopt the approach proposed in de Grancey et al. [2022], implemented in the PUNCC library Mendil et al. [2023], under the name *Split-Boxwise Conformal Object Detection*.

The principle of this method is to adjust each predicted bounding box using margins derived from a calibration set so that, with a given probability, the predicted box contains the true box.

Consider a ground-truth box  $Y$  characterized by its four coordinates :  $Y = (x^{\min}, y^{\min}, x^{\max}, y^{\max})$ . Similarly, each predicted box  $\hat{Y}_i$  is defined by the coordinates of its bottom-left and top-right corners.

**Nonconformity scores** For each correct prediction (a *True Positive*, i.e., matched to a ground truth according to a minimum IoU threshold), we compute a vector of nonconformity scores measuring the discrepancy between the matched prediction and ground truth. Unlike standard conformal prediction (where scores are scalars), here each box is represented by a four-dimensional vector :  $x_{\min}$ ,  $y_{\min}$ ,  $x_{\max}$  and  $y_{\max}$ . Two types of measures can be used :

- **Additive nonconformity score** : for each box  $k$  matched to a ground truth  $\mathbf{b}_k^{gt} = (x_{\min}^k, y_{\min}^k, x_{\max}^k, y_{\max}^k)$  and its prediction  $\hat{\mathbf{b}}_k = (\hat{x}_{\min}^k, \hat{y}_{\min}^k, \hat{x}_{\max}^k, \hat{y}_{\max}^k)$ , define :

$$\mathbf{r}_k^a = (\hat{x}_{\min}^k - x_{\min}^k, \hat{y}_{\min}^k - y_{\min}^k, \hat{x}_{\max}^k - x_{\max}^k, \hat{y}_{\max}^k - y_{\max}^k).$$

- **Multiplicative nonconformity score** : we can also normalize these discrepancies by the width  $\hat{w}$  and height  $\hat{h}$  of the predicted box :

$$\begin{aligned} \hat{w} &= \hat{x}_{\max} - \hat{x}_{\min}, & \hat{h} &= \hat{y}_{\max} - \hat{y}_{\min}, \\ \mathbf{r}_k^m &= \left( \frac{\hat{x}_{\min}^k - x_{\min}^k}{\hat{w}}, \frac{\hat{y}_{\min}^k - y_{\min}^k}{\hat{h}}, \frac{x_{\max}^k - \hat{x}_{\max}^k}{\hat{w}}, \frac{y_{\max}^k - \hat{y}_{\max}^k}{\hat{h}} \right). \end{aligned}$$

To establish matches between predictions and ground truth, we use the Kuhn–Munkres algorithm (or *Hungarian matching* Kuhn [1955]), here based on IoU with a 0.5 threshold.

**Bonferroni correction** Since the nonconformity vectors are four-dimensional, we obtain four distinct score distributions—one for each coordinate of the predicted box. To ensure joint coverage of at least  $1 - \alpha$  across the four dimensions, a Bonferroni correction is applied de Grancey et al. [2022].

To formally index these components, let  $j \in \{1, 2, 3, 4\}$  correspond respectively to :  $j = 1 : x_{\min}$ ,  $j = 2 : y_{\min}$ ,  $j = 3 : x_{\max}$ ,  $j = 4 : y_{\max}$ .

This amounts to computing, for each component  $j \in \{1, 2, 3, 4\}$ , a quantile at level  $\frac{\alpha}{4}$  :

$$\hat{q}(j) = q_{\lceil(1-\frac{\alpha}{4})(n+1)\rceil/n}(\{r_k(j) : k \in \{1, \dots, n\}\}),$$

where  $q_{\lceil(1-\frac{\alpha}{4})(n+1)\rceil/n}$  denotes the corrected quantile function,  $r_k(j)$  is the  $j$ -th component of the nonconformity vector  $r_k$ , and  $\hat{q}(j)$  is the associated empirical quantile. Here, index  $k$  runs over all  $n$  calibration samples and identifies, for each observation, the nonconformity value computed on component  $j$ .

These quantiles are then used to adjust predicted boxes at inference. For a prediction  $\hat{\mathbf{b}}_k$ , we build the *conformal* box  $\hat{\mathbf{b}}_k^{\text{conf}}$  by expanding the coordinates with margins derived from the quantiles  $\hat{q}(j)$ . This theoretically guarantees :

$$\mathbb{P}(\mathbf{b}_{n+1}^{gt} \subseteq \hat{\mathbf{b}}_{n+1}^{\text{conf}}) \geq 1 - \alpha,$$

where inclusion  $\mathbf{b} \subseteq \hat{\mathbf{b}}$  means :

$$x_{\min} \geq \hat{x}_{\min}^{\text{conf}}, \quad y_{\min} \geq \hat{y}_{\min}^{\text{conf}}, \quad x_{\max} \leq \hat{x}_{\max}^{\text{conf}}, \quad y_{\max} \leq \hat{y}_{\max}^{\text{conf}}.$$

In this framework, the goal is to extend the predicted box toward smaller values in the bottom-left (for  $x_{\min}$  and  $y_{\min}$ ) and toward larger values in the top-right (for  $x_{\max}$  and  $y_{\max}$ ), to ensure conservative coverage.

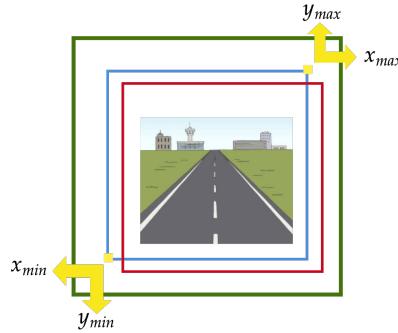


FIGURE 5.2 – Illustration of expanding the predicted box into a conformal box.

**Two expansion methods** The quantiles from each score distribution are then used at inference to adjust predictions by modifying the edges of each predicted box  $\hat{\mathbf{b}}_k$ . Recall that two approaches are possible for enlarging these boxes : an **additive** method and a **multiplicative** method.

In the **additive** case, the conformal box  $\hat{\mathbf{b}}_k^{\text{conf}}$  is obtained by directly adding or subtracting the quantiles to the predicted box coordinates. This amounts to expanding the box by fixed margins expressed in pixels :

$$\hat{\mathbf{b}}_k^{\text{conf}} = (\hat{x}_{\min}^k - \hat{q}_1, \hat{y}_{\min}^k - \hat{q}_2, \hat{x}_{\max}^k + \hat{q}_3, \hat{y}_{\max}^k + \hat{q}_4).$$



FIGURE 5.3 – Conformal boxes (in green) obtained with the additive method for YOLOv5 (left) and YOLOv6 (right).

In the **multiplicative** case, the same quantiles are now weighted by the dimensions of the predicted box. The margins are therefore proportional to the size of the detected object, allowing the method to adapt dynamically to different object scales :

$$\hat{\mathbf{b}}_k^{\text{conf}} = (\hat{x}_{\min}^k - \hat{q}_1 \hat{w}, \hat{y}_{\min}^k - \hat{q}_2 \hat{h}, \hat{x}_{\max}^k + \hat{q}_3 \hat{w}, \hat{y}_{\max}^k + \hat{q}_4 \hat{h}),$$

where  $\hat{w}$  and  $\hat{h}$  denote respectively the width and height of the predicted box.



FIGURE 5.4 – Conformal boxes (in green) obtained with the multiplicative method for YOLOv5 (left) and YOLOv6 (right).

### Algorithm summary

1. Train an object detection model (e.g., YOLO).
2. Build a calibration set, distinct from the test set.
3. Identify correct predictions (*True Positives*) using an IoU threshold, then match predicted boxes to ground-truth boxes using the Hungarian algorithm.
4. Compute nonconformity vectors for each valid match.
5. Estimate Bonferroni-corrected quantiles for each component of the nonconformity vectors.
6. Construct conformal boxes at inference by enlarging predictions with margins derived from the quantiles.

This method thus yields, for each prediction, a box augmented with a probabilistic coverage guarantee, providing a statistically grounded framing of spatial uncertainty in object detection.

### 5.3 How to evaluate a robust detection ?

By definition, a robust detection is characterized by the presence of at least three keypoints usable for the pose-estimation problem, better known as **PnP** (Perspective-n-Point, or P3P in the minimal three-point case), and thus by sufficient coverage of the runway (or region of interest).

To formalize this notion, we introduce the **coverage** metric, a deterministic measure indicating whether the detection sufficiently covers the region of interest. More precisely, *coverage* is defined as a binary function equal to 1 if the ground-truth box is entirely covered by the predicted box, and 0 otherwise :

$$\text{coverage}(b_{\text{pred}}, b_{\text{gt}}) = \begin{cases} 1 & \text{if } b_{\text{gt}} \subseteq b_{\text{pred}} \\ 0 & \text{otherwise} \end{cases}$$

where  $b_{\text{pred}}$  is the box predicted by the detector, and  $b_{\text{gt}}$  the ground-truth box.

This notion is intrinsically related to **Intersection over Area (IoA)**, which measures the fraction of the ground truth covered by the detection, as defined previously.

Thus, the binary coverage metric can be seen as a strict version of IoA with a fixed threshold at 1.

This metric provides an initial indication of how the base predictor covers the ground truth and whether it meets the minimal robustness criterion. However, it is pertinent to go further and ask whether considering only coverage of the ground truth is sufficient to evaluate detection robustness.

*Coverage* can be illustrated as follows :

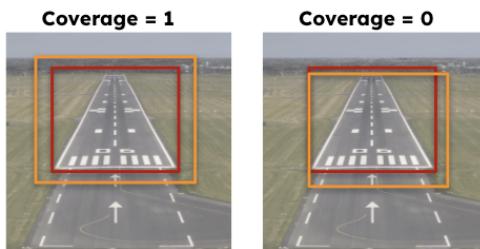


FIGURE 5.5 – Illustration of the *coverage* metric : the ground truth (in red) is fully included in the conformal box (in orange).

#### 5.3.1 Are IoU and IoA robust metrics ?

In VLS Phase 1, it is crucial to identify the runway region precisely. The robustness of detection therefore rests on two complementary requirements : full coverage of the ground truth and good spatial alignment. These

constraints raise the question of whether classical metrics such as IoU and IoA are sufficient to effectively assess robustness.

IoU is the standard metric for measuring overlap between the predicted box  $\mathbf{b}_{\text{pred}}$  and the true box  $\mathbf{b}_{\text{gt}}$ . It reflects the level of overlap and alignment between the two boxes : the higher the IoU, the better the localization.

IoA, for its part, measures the fraction of the ground truth covered by the prediction. It is therefore directly linked to the requirement of full coverage, indispensable for ensuring visibility of at least three runway corners—the minimum condition for pose estimation.

However, neither of these two metrics alone suffices to guarantee a robust detection. IoA equal to 1 can easily be obtained by detecting a very large region, or even the entire image, without respecting the necessary alignment (thus with low IoU). An illustrative example is shown in Figure 5.6, which compares a trivial-yet-imprecise coverage case with a truly precise, well-fitted detection.

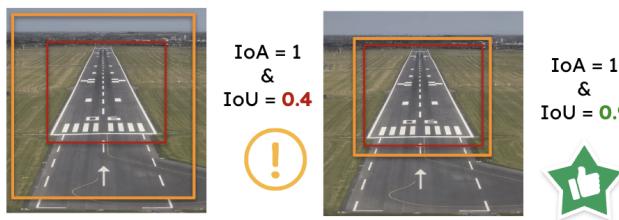


FIGURE 5.6 – Comparison of two cases : left, the ground truth is covered (IoA = 1) but poorly localized (low IoU) ; right, both metrics are satisfied.

A robust detection can thus be defined only by the simultaneous satisfaction of these two criteria :

$$\text{Robust detection} \iff \text{IoU}(\mathbf{b}_{\text{pred}}, \mathbf{b}_{\text{gt}}) \geq t \quad \text{and} \quad \text{IoA}(\mathbf{b}_{\text{pred}}, \mathbf{b}_{\text{gt}}) = 1,$$

where  $t$  is a minimum IoU threshold ensuring sufficiently precise localization.

Figures 1.6b and 1.6a illustrate this complementarity on concrete pipeline examples, showing that only the combination of the two metrics effectively judges robustness.

### 5.3.2 C-mAP : Conformal Mean Average Precision

The above definition of a robust detection reveals a double requirement that goes beyond using IoU alone as an evaluation criterion. This double constraint—full coverage and spatial precision—calls for a suitable metric, particularly when evaluating *conformal predictors*, which are constrained to guarantee that each detected box fully covers the ground truth (IoA = 1).

For a standard predictor such as YOLO, performance is usually evaluated only via IoU. That does not measure the impact of adding a strict coverage constraint on the ability to maintain precise localization. This situation is summarized in the following table :

Method	$\text{IoU} \geq t$	$\text{IoA} = 1$	Robust ?
YOLO	✓	✗	✗
Conformalized-YOLO	?	✓	?

To address this, we introduce the **C-mAP** (*Conformal mean Average Precision*) metric, which evaluates performance while simultaneously imposing :

- a minimum IoU threshold ( $\text{IoU} \geq t$ ) guaranteeing spatial precision,
- a strict coverage constraint ( $\text{IoA} = 1$ ).

This metric makes it possible to evaluate the trade-off a conformal detector must make to satisfy both full coverage and spatial alignment. In practice, it quantifies to what extent the conformity constraint degrades detection

quality in terms of IoU. You can find a Python implementation of this function in Appendix .2. This metric will be used to compare the performance of classical and conformal models in the experiments section.

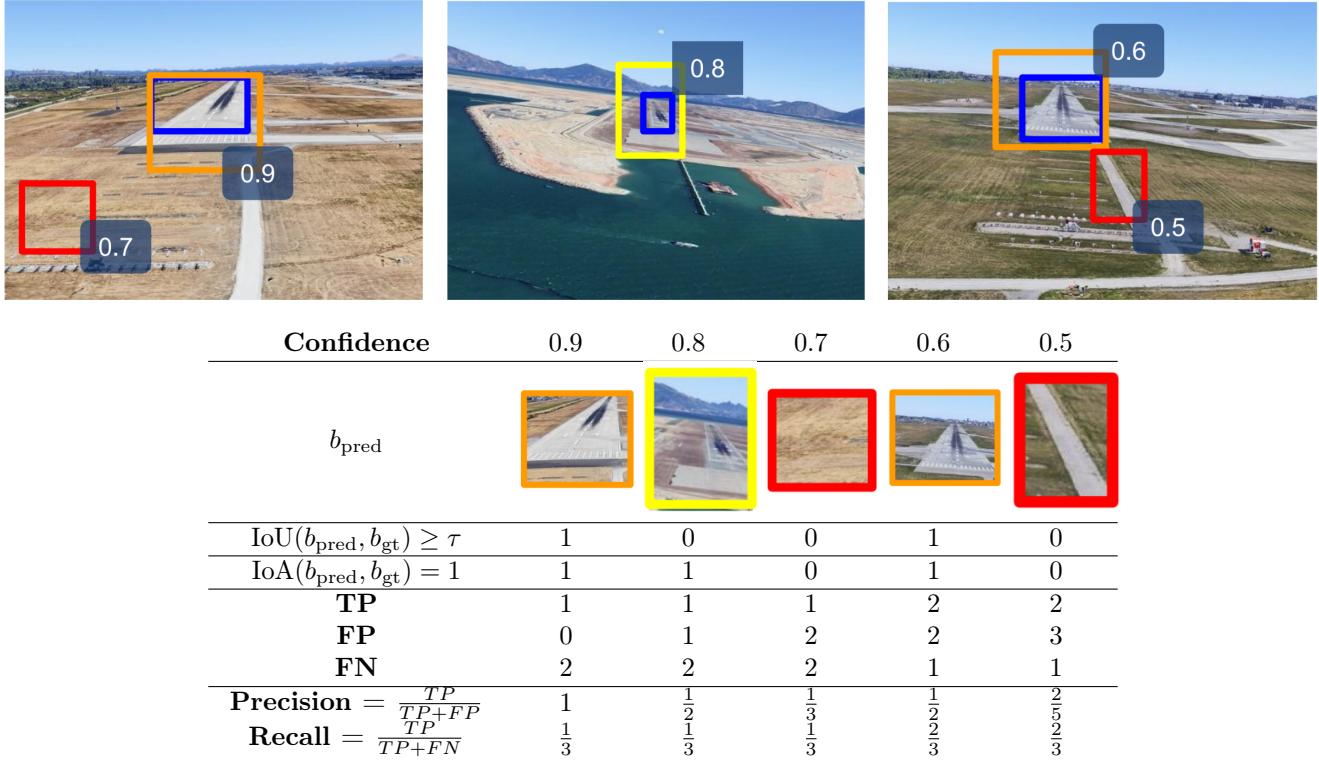


FIGURE 5.7 – Computation of **C-(m)AP** step-by-step on an example from the LARD dataset. The top image shows the visual setup, while the table illustrates how predictions at different confidence levels affect precision and recall.

### 5.3.3 Other evaluation metrics

**Correction margins** To measure the effect of conformalization on initial predictions, we introduce the notion of *margin*, which quantifies the pixel difference between a predicted box and its conformalized version. For each direction  $d \in \{\text{left}, \text{right}, \text{top}, \text{bottom}\}$ , the average margin is computed as follows :

$$\text{Margin}(d) = \frac{1}{n} \sum_{i=1}^n \left| \hat{\mathbf{b}}_{i,d}^{\text{conf}} - \hat{\mathbf{b}}_{i,d} \right|,$$

where  $\hat{\mathbf{b}}_{i,d}$  and  $\hat{\mathbf{b}}_{i,d}^{\text{conf}}$  represent respectively the  $d$  coordinate of the predicted box and its conformalized version for example  $i$ , and  $n$  is the total number of predictions. Large margins indicate a substantial expansion, while small margins indicate a minor correction. Balanced margins in all directions suggest a symmetric modification.

**Expansion rate (Stretch)** To complement this analysis, we also use the *Stretch* metric introduced in Andéol et al. [2023], which measures the expansion ratio between the conformalized box and the initially predicted box in terms of area :

$$\text{Stretch} = \frac{1}{n} \sum_{i=1}^n \sqrt{\frac{\text{Area}(C_i)}{\text{Area}(P_i)}}$$

where  $\text{Area}(C_i)$  is the area of conformalized box  $C_i$  and  $\text{Area}(P_i)$  that of predicted box  $P_i$ .

A *Stretch* value close to 1 indicates that the predicted box was little modified, reflecting an initially precise prediction and thus high model confidence. Conversely, a value significantly greater than 1 reflects noticeable expansion, typically in response to an initially too-narrow prediction to cover the object with certainty. This reveals model uncertainty about the exact object boundaries.

## 5.4 Experiments

### 5.4.1 Data and model training

From the LARD dataset Ducoffe et al. [2023], we created three subsets : a training set, a validation set, and a test set. The base predictor used to compute mAP is trained on the training set and evaluated on the validation set. The completely independent test set is reserved for conformal prediction experiments. It is split into two parts : 80% of the images are used for calibration, and the remaining 20% are used to evaluate the conformal predictors via the *Stretch*, *Coverage* and *C-mAP* metrics.

Table 5.1 presents the distribution of images by type and set. Recall that LARD contains both synthetic and real images ; however, only real images are kept in the test set.

The current dataset version has only one runway annotation per image. Nevertheless, our method is already designed to adapt to multiple-runway detection if such annotations become available in future versions.

Set	Type	Number of images
Training	Synthetic	11 546
Validation	Synthetic	2 886
Test	Real + Synthetic	2 315

TABLE 5.1 – Dataset split by set, image type, and total count.

### 5.4.2 Model configurations and training

In this section, we present the models used prior to conformalization. Two pretrained models from the YOLO (You Only Look Once) family were chosen for their good trade-off between detection accuracy and computational efficiency : **YOLOv5-small** Jocher [2020] and **YOLOv6-small** Li et al. [2023].

Both models were initialized from pretrained weights available in their respective official repositories, then finetuned on our dataset.

- **YOLOv5-small** was trained for 100 epochs with a batch size of 16 and an input resolution of  $640 \times 640$  pixels. This widely used configuration in the YOLO community enabled good convergence.
- **YOLOv6-small** was trained for 97 epochs with a batch size of 32, keeping the same  $640 \times 640$  input resolution. Training was conducted via the official YOLOv6 repository with default optimization parameters.

In both cases, training includes advanced data augmentation techniques, essential for good generalization to aerial images with significant variations in lighting, weather, and perspective. Augmentation follows the default configurations provided in each repository.

A comparative performance summary of the two models is presented in Table 5.2, based on results obtained on the validation set.

### 5.4.3 Results

All results presented in this section were obtained with a risk level set to  $\alpha = 0.3$ . Recall that we use the *PUNCC* library Mendil et al. [2023] to apply conformal prediction to YOLOv5 and YOLOv6, yielding four conformal variants : c-YOLOv5-a and c-YOLOv6-a for the additive-penalty version, c-YOLOv5-m and c-YOLOv6-m for the

TABLE 5.2 – Comparison of pretrained object detectors on the runway detection task, evaluated on the validation set.

Metric	YOLOv5 (pretrained)	YOLOv6 (pretrained)
mAP@0.5	0.995	0.9901
mAP@0.5 :0.95	0.9712	0.9413
GFLOPs	15.8	45.3

multiplicative-penalty version. The additive penalty applies a fixed enlargement of the boxes, while the multiplicative penalty adapts the expansion to the size of the base predicted box.

Table 5.3 presents the average magnitude of the margins added on the four sides of the predicted boxes to obtain conformal boxes. We observe that additive-penalty models generally produce smaller and more balanced margins, in particular c-YOLOv6-a, which shows the smallest margins and the lowest overall mean (7.03 pixels), indicating good confidence of the base model. Conversely, c-YOLOv5-m requires larger corrections, especially on the left and right sides (up to 18 pixels), reflecting greater uncertainty in initial predictions.

Model	Left	Top	Right	Bottom	Overall mean
c-YOLOv5-a	11.76	5.79	8.93	8.54	8.76
c-YOLOv5-m	18.17	7.85	12.04	10.65	12.18
<b>c-YOLOv6-a</b>	<b>10.39</b>	<b>3.47</b>	<b>7.76</b>	<b>6.52</b>	<b>7.03</b>
c-YOLOv6-m	15.58	4.92	11.75	7.32	9.89

TABLE 5.3 – Average predictive margins (in pixels) for each side of the conformal boxes.

Table 5.4 then shows the average conformal-box expansion (*Stretch*), the square root of their area, and the coverage rate (*Coverage*), measuring the proportion of predictions that fully cover the ground truth. Note that even if the nominal risk level is set to  $\alpha = 0.3$ , the empirical coverage values observed are slightly higher (up to 77.06%). This is not problematic : it stems from the conservative nature of conformal prediction, and more specifically from the use of the Bonferroni correction, which slightly enlarges quantiles to guarantee simultaneous coverage over the four box coordinates. Thus, even if conformal boxes are sometimes larger, this slight expansion remains controlled. It does not negatively impact spatial precision : the high C-mAP scores confirm that predicted regions remain well aligned with the ground truth, without causing misalignments or interpolation errors. In practice, this maintains good localization of key runway elements while offering reliable coverage guarantees. Model c-YOLOv5-a achieves the highest coverage (77.06%), but c-YOLOv6-m obtains the best Stretch score (1.11), suggesting more precise boxes requiring less expansion. In terms of area, c-YOLOv6-a produces the most compact and regular boxes, with low variance (156.60), which indicates stable predictions.

Model	Stretch	$\sqrt{\text{Area}}$	Coverage (%)
c-YOLOv5-a	1.16	$194.92 \pm 156.74$	<b>77.06</b>
c-YOLOv5-m	1.13	$200.42 \pm 177.45$	75.88
c-YOLOv6-a	1.13	<b><math>188.76 \pm 156.60</math></b>	75.73
c-YOLOv6-m	<b>1.11</b>	$193.05 \pm 173.47$	73.93

TABLE 5.4 – Analysis of conformal boxes : expansion, average area, and coverage rate.

Finally, Table 5.5 evaluates the models' precision according to three criteria : standard mAP (based only on IoU), C-mAP (imposing IoA = 1 in addition to IoU), and C-mAP@50@80 :100, which introduces a progressive scale of IoA thresholds between 0.8 and 1.0. Classical models (YOLOv5 and YOLOv6) retain very high mAP scores (96.88 and 98.13 respectively), but drop drastically on C-mAP (less than 2%), confirming that they almost never guarantee full coverage. Conformal versions, on the other hand, display much higher C-mAP scores (between 52.71% and 56.86%), while maintaining mAP above 92%, which shows that the loss in precision is limited despite added

constraints. The C-mAP@50@80 :100 metric follows the same trend and indicates that conformal models remain robust even as the coverage requirement becomes more stringent.

Model	mAP	C-mAP	C-mAP@50@80 :100
YOLOv5	96.88	0.77	46.92
c-YOLOv5-a	92.67	<b>56.86</b>	80.73
c-YOLOv5-m	96.17	55.84	<b>82.18</b>
YOLOv6	<b>98.13</b>	1.31	51.94
c-YOLOv6-a	95.09	55.75	81.86
c-YOLOv6-m	96.71	52.71	81.93

TABLE 5.5 – Performance of YOLO and conformal versions on the overall test set.

These results confirm the effectiveness of conformal prediction for adapting object detectors to stricter criteria, particularly where full coverage is indispensable. The proposed C-mAP metric proves more relevant for evaluating model robustness in demanding operational scenarios, combining spatial precision (via IoU) with coverage guarantees (via IoA).

## 5.5 Conclusion

Our work highlighted certain limits of object-detection algorithms, especially their ability to produce predictions that are correct from a classification standpoint but imprecise in terms of localization—particularly regarding overlap with ground truth.

We demonstrated the importance of quantifying the risk associated with such predictions and showed that using metrics like IoA, especially under the strict condition  $\text{IoA} = 1$ , can provide a reliable coverage guarantee. This constraint could, in time, be leveraged as an assurance criterion in critical applications.

Our current work is limited to risk evaluation on pretrained predictors on the LARD dataset, notably due to conformal prediction’s incompatibility with temporally correlated image streams. However, these results open the door to considering the potential benefit of integrating coverage guarantees—such as the  $\text{IoA} = 1$  constraint—already during the training phase of detection models.

Ultimately, quantifying uncertainty helps better understand the risks inherent in using these detections within a complete VLS pipeline. This lays the groundwork for developing a differentiable *end-to-end* pipeline capable of performing all VLS steps in an integrated and reliable manner.

## Chapitre 6

# POSEIDON : POSe Estimation with Explicit Differentiable OptimizatioN

### Sommaire

---

6.1	Context and Objectives . . . . .	43
6.2	Clarification on the Term <i>Pose</i> . . . . .	44
6.3	Pose-Oriented Detection with YOLO-NAS-POSE . . . . .	45
6.3.1	Architecture Overview . . . . .	45
6.3.2	Neural Architecture Search (NAS) . . . . .	48
6.3.3	Quantization-Aware Training (QAT) . . . . .	48
6.3.4	Data Format and Preparation for YOLO-NAS-POSE . . . . .	49
6.3.5	Losses . . . . .	50
6.3.6	Fine-Tuning YOLO-NAS-POSE on LARD . . . . .	51
6.4	Toward an End-to-End Pipeline for Pose Estimation . . . . .	53

---

### 6.1 Context and Objectives

In this project, we focus on the final task of the VLS (Vision Landing System) : estimating the pose of the onboard camera, i.e., its position and orientation in space. This step is crucial to precisely guide the aircraft toward the runway. Indeed, an accurate estimate of the camera's position (and thus the aircraft's) strengthens learning by steering supervision toward key regions of the image, called *aiming points*.

The problem can be formulated as follows : given the intrinsic parameters of a pinhole camera (matrix  $A$ ) and a set of correspondences between 3D points in the real world and their 2D projections in the image, we seek to estimate the camera pose, represented by a rotation  $R$  and a translation  $C$ .

Recall that the VLS pipeline decomposes into three steps :

1. Runway detection.
2. Extraction of the four runway corners in a 2D image.
3. Camera pose estimation via a geometric optimization algorithm (typically a PnP method).

I present here our project **POSEIDON** (*POSe Estimation with Explicit Differentiable OptimizatioN*) where we explore the feasibility of a more integrated, *towards end-to-end* architecture aimed at eliminating post-processing steps while taking into account operational safety constraints.

I explore this more integrated approach using **YOLO-NAS-POSE** by DeciAI, which combines object detection and 2D keypoint regression. This architecture, optimized via *Neural Architecture Search* and compatible with quantization, is suited to embedded constraints where inference time and energy efficiency are critical.

We consider two strategies :

- **Partially end-to-end** : use YOLO-NAS-POSE to detect the runway and its corners, followed by pose estimation via OpenCV (PnP method).
- **Fully end-to-end** : directly integrate pose estimation into training via a loss function.

The PnP problem, notably in its **P3P** variant, offers fast solutions but is not coded in a differentiable framework such as PyTorch, preventing its direct integration into a network. To circumvent this limitation, we use the formulation proposed by Kneip et al. [2011] and implement it in PyTorch to enable backpropagation and integration into YOLO-NAS-POSE.

This chapter therefore aims to lay the theoretical and practical foundations of the POSEIDON project. I seek to :

- Assess the feasibility of an end-to-end VLS pipeline.
- Exploit camera geometry to strengthen supervision (*pose-aware loss*).
- Develop a differentiable and optimized version of P3P.

The main challenge here is to improve robustness and accuracy of detection in critical situations, while ensuring an inference time compatible with embedded deployment constraints.

## 6.2 Clarification on the Term *Pose*

In the literature, the term *pose* is polysemous and can be confusing, as it covers two distinct meanings depending on the context :

**1. Meaning in 2D detection (image-based vision)** In this first sense, widely used in *pose estimation* algorithms applied to object or human detection, *pose* denotes a set of keypoints localized in 2D coordinates in the image. These points, obtained by direct regression, visually characterize the relative position of object parts in the image plane. For example, in the case of a human body, the *pose* may correspond to the 2D positions of joints (shoulders, elbows, knees, etc.) detected in the image. In the **YOLO-NAS-POSE** setting, we speak of *2D pose* because the network predicts the  $(u, v)$  coordinates of the *keypoints* directly in the image frame :

$$\mathbf{k}_i^{2D} = (u_i, v_i), \quad i = 1, \dots, N.$$

**2. Geometric meaning (position and orientation in 3D)** In a second, more geometric sense—common in robotics or 3D computer vision—*pose* denotes the full position and orientation of an object or a camera in a 3D frame. It is typically represented by :

- a rotation matrix  $\mathbf{R} \in SO(3)$  describing orientation,
- a translation vector  $\mathbf{C} \in \mathbb{R}^3$  representing the position of the camera center in the world frame.

The perspective projection matrix  $\mathbf{P}$  linking world 3D points  $\mathbf{P}_i$  to their 2D projections  $\mathbf{p}_i$  is then written :

$$\mathbf{p}_i \sim \mathbf{A} [\mathbf{R} \mid \mathbf{t}] \mathbf{P}_i,$$

where  $\mathbf{A}$  is the camera intrinsic matrix and  $\mathbf{t} = -\mathbf{R} \mathbf{C}$ . **Example :** Consider a camera located 6 meters in front of the runway, oriented to look straight ahead. The rotation  $\mathbf{R}$  can be the identity matrix and the position  $C$  :

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 \\ 0 \\ 6 \end{bmatrix}.$$

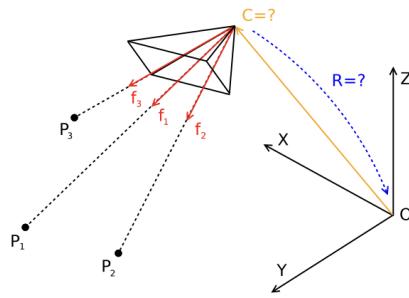


FIGURE 6.1 – Illustration of the pose estimation problem from Kneip et al. [2011] (P3P)

In **YOLO-NAS-POSE**, the term *pose* refers to the first meaning (detection of 2D *keypoints*). However, when we address the P3P problem, our use of the term *pose* refers to the second meaning : it is a matter of determining the camera position **C** and orientation **R** in the world frame from correspondences between 3D points and their 2D projections, given the camera intrinsics. Thus, in our POSEIDON pipeline, *2D pose* serves as input to estimate the camera's *3D pose* via a differentiable geometric optimization step.

### 6.3 Pose-Oriented Detection with YOLO-NAS-POSE

YOLO-NAS-Pose is a model developed by Deci AI that uses *Neural Architecture Search* (NAS) to generate architectures optimized for speed and accuracy. Unlike general-purpose detection models, YOLO-NAS-Pose is pre-trained on the COCO dataset for a specific task : people detection and human pose estimation. It is deliberately limited to a single “human” class and predicts, for each person, 17 keypoints corresponding to the main body joints (head, shoulders, elbows, wrists, hips, knees, ankles, etc.).

In the VLS context, this type of model is of great interest because it combines lightness and inference speed.

In this section, we successively present the key elements of this architecture :

- the principle of NAS used to design the model,
- the structure of the optimized *backbone* for visual feature extraction,
- the specificity of quantization-aware blocks (*Quantization-Aware Training*),
- the training dataset formatting and the definition of the different loss functions used to train the model,
- and finally the fine-tuning of YOLO-NAS-POSE on the LARD dataset and its results.

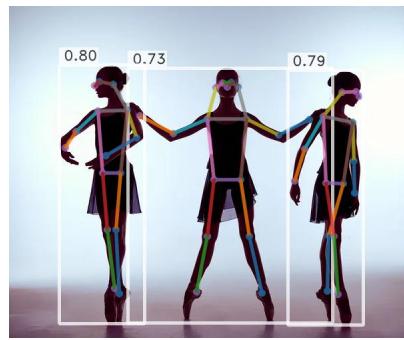


FIGURE 6.2 – Example prediction on an image of dancers

#### 6.3.1 Architecture Overview

The YOLO-NAS architecture, and by extension YOLO-NAS-Pose, follows a three-part structure typical of object detectors : *Backbone*, *Neck*, and *Head*. It incorporates several key elements from YOLO variants presented previously

(see 3.3), such as non-maximum suppression (NMS) and *Average Precision* computation, used to optimize prediction quality. This design enables efficient visual feature extraction, multi-scale processing, and simultaneous prediction of multiple outputs (bounding boxes, classes, and keypoints). The main objective is to perform object detection and keypoint estimation in a single pass, thereby optimizing the trade-off between accuracy and speed, which is particularly suited to embedded or real-time contexts.

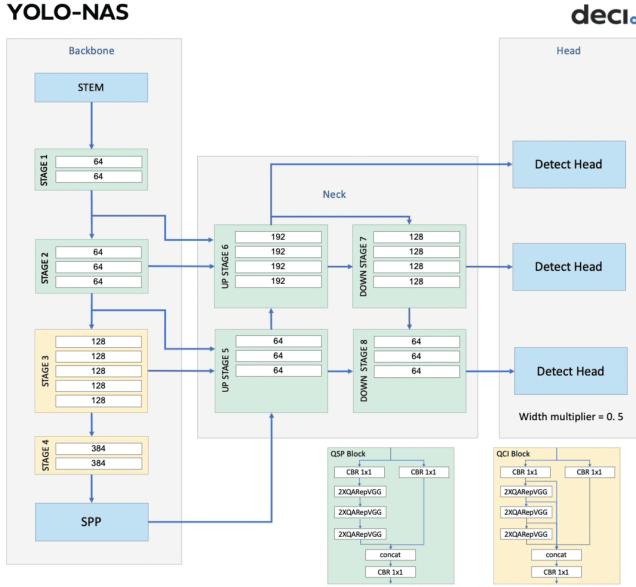


FIGURE 6.3 – YOLO-NAS Backbone architecture

<https://learnopencv.com/yolo-nas-pose/>

**Backbone — feature extraction** The *backbone* extracts relevant visual descriptors from the input image. The image  $\mathbf{I} \in \mathbb{R}^{H \times W \times 3}$  is first processed by an initial block (*STEM*) composed of convolutional layers to slightly reduce the resolution while increasing channel richness. The information then goes through several *stages* (STAGE 1 to STAGE 4), each formed by optimized convolutional blocks (e.g., *QARepVGG*, discussed later). At each transition to the next stage, a *downsampling* operation (by stride-2 convolution or pooling) halves the height and width, while the number of channels increases to represent more complex features. For example :

$$640 \times 640 \times 3 \longrightarrow 320 \times 320 \times 64 \longrightarrow 40 \times 40 \times 384$$

Thus, spatial resolution decreases, but each “pixel” of the feature map contains richer, more abstract information. At the end of the *backbone*, a *Spatial Pyramid Pooling* (SPP) block combines information from different spatial scales, improving robustness to object size variations.

**Neck — multi-scale fusion** The *neck* bridges the features extracted by the *backbone* and the final prediction layers. Its role is to combine information from different spatial scales so that the network can detect both small and large objects.

To do this, it adopts a *Feature Pyramid Network* (FPN) structure implementing two complementary operations :

- **Up-sampling** : increase the spatial resolution of a feature map (e.g., from  $40 \times 40$  to  $80 \times 80$ ), via interpolation (bilinear, bicubic) or transposed convolutions. The goal is to “bring up” information extracted by deep layers (semantically rich but spatially coarse) to a finer resolution, improving the localization of small objects.
- **Down-sampling** : conversely, reduce spatial resolution (e.g., from  $80 \times 80$  to  $40 \times 40$ ) via stride  $> 1$  convolutions or *pooling*. This simplifies information and emphasizes global context, useful for detecting large

objects.

Multi-scale fusion consists of combining (often by concatenation) the maps obtained after up-sampling and down-sampling, followed by convolutions to mix and refine them. This step yields tensors containing both local detail and a global understanding of the scene—essential for robust detection across object sizes.

**Head — final predictions** The YOLO-NAS head predicts bounding boxes, objectness scores, and classes at three scales corresponding to strides  $s \in \{8, 16, 32\}$ . For each grid cell and *anchor*, the output is a vector of dimension :

$$\mathbf{y} \in \mathbb{R}^{4+1+C},$$

where 4 corresponds to bounding box coordinates  $(x, y, w, h)$ , 1 to the objectness score, and  $C$  to the number of classes, as in basic YOLO.

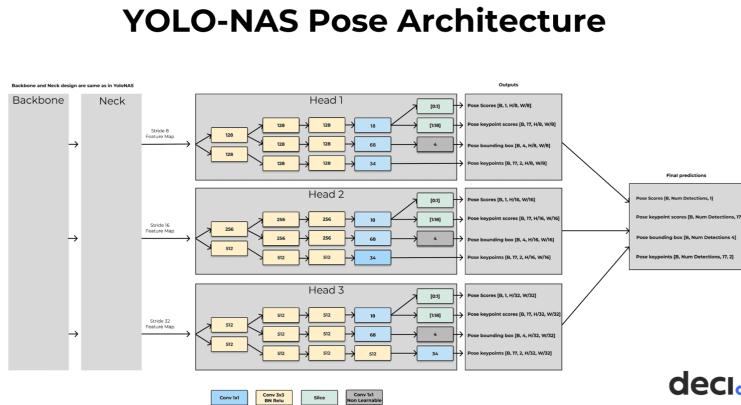


FIGURE 6.4 – YOLO-NAS-POSE extension

<https://learnopencv.com/yolo-nas-pose/>

**YOLO-NAS-Pose extension** YOLO-NAS-Pose reuses the same *backbone* and *neck*, but modifies the *head* to include 2D pose (keypoint) estimation. This head predicts, still at three scales, an enriched vector :

$$\mathbf{y}_{pose} \in \mathbb{R}^{1+4+(K \times 2)+K},$$

where :

- 1 : score for the presence of a pose (analogous to objectness) ;
- 4 : bounding box coordinates  $(x, y, w, h)$  ;
- $K \times 2$  :  $(x, y)$  coordinates of the  $K$  keypoints (with  $K = 17$  for the human body, i.e., 34 values) ;
- $K$  : visibility scores  $v_i$  associated with each keypoint.

Thus, for  $K = 17$  :

$$\dim(\mathbf{y}_{pose}) = 1 + 4 + 34 + 17 = 56.$$

Each prediction is made at all grid positions, enabling simultaneous detection of the object's position and the spatial configuration of its keypoints. The main challenge is to fine-tune this COCO-pretrained architecture on the LARD dataset by restricting detection to the **4 keypoints** of interest.

### 6.3.2 Neural Architecture Search (NAS)

*Neural Architecture Search* (NAS) is an *AutoML* technique that automates the design of neural architectures. Rather than relying on human expertise and time-consuming design–experiment cycles, NAS explores a large **search space**—defined by possible configurations of convolutional layers, activation functions, residual connections, etc.—to select the best-performing models for a given task.

The search can be driven by different **strategies** :

- **Reinforcement learning** : an agent composes architectures and receives a reward based on performance.
- **Evolutionary algorithms** : architectures evolve via mutations and crossovers ; only the best survive.
- **Random search** : simple but effective to establish an initial performance–resource frontier.

Candidate **evaluation** is often costly : training each architecture to convergence is prohibitive. Methods like Deci.ai’s **AutoNAC** use **proxy tasks** and **early stopping** to quickly estimate architecture quality. AutoNAC goes further by integrating *target hardware awareness* (CPU, GPU, edge devices) into optimization, producing faster and leaner models while maintaining accuracy.

NAS typically adopts **multi-objective optimization** : beyond accuracy, it considers latency, memory, and compute cost—ideal for embedded deployments.

### 6.3.3 Quantization-Aware Training (QAT)

A major objective of our project is to perform detection and pose estimation in real-time on an embedded system with limited resources. To achieve this, **YOLO-NAS-POSE** integrates *Quantization-Aware Training* (QAT), which consists in training the network while simulating the conversion of its weights and activations to a more compact format, typically 8-bit integers (INT8).

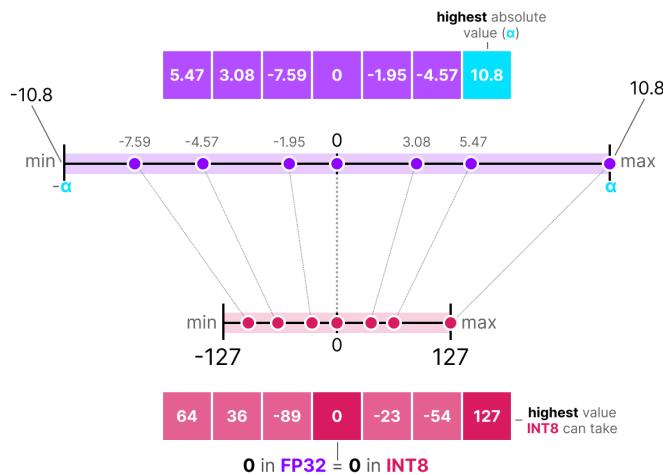


FIGURE 6.5 – Linear mapping of FP32 values to INT8 via symmetric quantization.

In standard training, weights and activations are represented in 32-bit floating-point precision (FP32), offering a broad value range and fine granularity (precision on the order of  $10^{-7}$ ). INT8 quantization reduces this range : values can only take 256 discrete levels (from  $-127$  to  $127$  for the signed format). To map from the continuous (FP32) to this discrete (INT8) domain, we apply a **linear mapping** :

$$v_{\text{INT8}} = \text{round} \left( \frac{v_{\text{FP32}}}{\alpha} \times 127 \right), \quad v_{\text{FP32}} \approx \frac{\alpha}{127} \times v_{\text{INT8}},$$

where  $\alpha$  is the absolute maximum observed in the data (weights or activations) and serves as a *scale factor*.

Intuitively, we can view this as “compressing” a ruler graded in millimeters (FP32) 6.5 into one graded in centimeters (INT8). Extreme FP32 values align with the INT8 bounds  $-127$  and  $127$ , and all other intermediate values are projected proportionally. This reduces representational granularity : small variations are lost because multiple distinct FP32 values round to the same INT8 value.

When applied bluntly after training (PTQ, *Post-Training Quantization*), this compression can severely degrade model accuracy. By contrast, QAT introduces this mapping during training : at each iteration, the network “sees” its weights and activations passed through this transformation and learns to compensate for the lost granularity. Weights are still stored and updated in FP32 to ensure optimization stability.

In our case, this controlled quantization considerably reduces memory footprint and inference latency, while preserving sufficient accuracy for runway detection and 2D pose estimation.

#### 6.3.4 Data Format and Preparation for YOLO-NAS-POSE

To train a pose detection model with **YOLO-NAS Pose**, annotations must be structured according to a modified **COCO keypoint** format. This format includes, for each image, the localization of *keypoints* as well as a *bounding box* covering the object of interest.

In our case, each image is associated with four keypoints denoted  $A$ ,  $B$ ,  $C$ , and  $D$ , typically representing the four corners of a runway. The points are connected in the order  $[A \rightarrow B \rightarrow C \rightarrow D \rightarrow A]$ , forming a closed loop (described by the `skeleton` field).

Annotations are extracted from `.csv` files containing keypoint coordinates and aggregated into a JSON structure compatible with the format expected by YOLO-NAS Pose. The process follows these steps.

---

**Algorithme 1 :** Generating the COCO-format annotation file for YOLO-NAS Pose

---

**Data :** Folder containing CSV files, images `train/`, `val/`, `test/`

**Result :** JSON file containing COCO-format annotations

- 1 Initialize `json_data` with fields `info`, `categories`, `images`, `annotations`
  - 2 **foreach** `.csv` file in the `CSV` folder **do**
  - 3     Read each line
  - 4     Extract the image name and coordinates  $(x_A, y_A), \dots, (x_D, y_D)$
  - 5     Index annotations by image name
  - 6 **foreach** `image` in `train/`, `val/`, `test/` **do**
  - 7     **if** corresponding annotation is missing **then**
  - 8         Skip the image
  - 9     Extract image dimensions (`width`, `height`)
  - 10    Extract keypoint coordinates  $A, B, C, D$
  - 11    Build the keypoint list :  $[x_A, y_A, 2, \dots, x_D, y_D, 2]$
  - 12    Compute the `bbox` enclosing the 4 points
  - 13    Add to `images` : `file_name`, `id`, `width`, `height`
  - 14    Add to `annotations` : `id`, `image_id`, `category_id`, `bbox`, `keypoints`, `num_keypoints`
  - 15    Increment image and annotation identifiers
  - 16 Save `json_data` to `yolonas_pose_annotations.json`
-

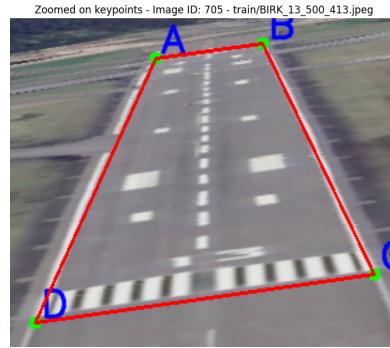


FIGURE 6.6 – Labeling

### 6.3.5 Losses

The *YOLO-NAS-Pose* model is designed to learn two complementary tasks simultaneously :

- **Object detection** — localize bounding boxes corresponding to people in the image.
- **2D pose estimation** — predict keypoints (joints) within each detected bounding box.

Before computing the loss for these tasks, we must match model predictions (boxes, scores, and poses) with ground-truth annotations. This matching is performed by `YoloNASPoseTaskAlignedAssigner`, which produces a data structure containing assigned correspondences. This structure is then used by `YoloNASPoseLoss` for the actual loss computation.

The matching process unfolds in several steps :

1. Compute an alignment metric between each predicted box (across all levels of the feature pyramid) and each ground-truth box.
2. Select the *top-k* most promising predicted boxes for each ground-truth box using IoU (for bounding boxes) and OKS (for keypoints).
3. Restrict positive samples to boxes whose center lies inside the ground-truth box.
4. If a predicted box is assigned to multiple ground truths, only the one with the best IoU (Intersection over Union) is kept.

IoU is computed between ground-truth and predicted boxes. If the argument `multiply_by_pose_oks` is *True*, this measure is weighted by the similarity of keypoint positions via the Object Keypoint Similarity (OKS), which quantifies how well predicted keypoints match ground-truth ones. This refines matching by integrating pose-specific information.

These elements then serve as the basis for the global model loss.

**Object Keypoint Similarity (OKS)** *Object Keypoint Similarity* (OKS) is a metric used to evaluate 2D pose estimation accuracy between two sets of keypoints : model predictions and ground truth. It plays an important role in matching predicted boxes to annotations, complementing the IoU score.

The implemented OKS formula is :

$$\text{OKS} = \frac{\sum_i \exp\left(-\frac{d_i^2}{2s^2\kappa_i^2}\right) \cdot \delta(v_i > 0)}{\sum_i \delta(v_i > 0)}$$

where :

- $d_i$  is the Euclidean distance between the predicted and ground-truth keypoint for joint  $i$ ,
- $s$  is the class scale (generally computed as the square root of the bounding box area),
- $\kappa_i$  is a per-joint constant representing acceptable error tolerance,

- $v_i$  is a visibility indicator for the ground-truth keypoint ( $v_i > 0$  means the point is visible),
- $\delta(v_i > 0)$  is an indicator function equal to 1 if  $v_i > 0$ , 0 otherwise.

OKS thus weights each keypoint's contribution by its visibility, localization precision, and object scale. In YOLO-NAS-Pose, this measure adjusts matching scores during assignment of predictions to ground truth. By multiplying IoU by OKS, the model favors predictions whose keypoints are correctly localized, improving the robustness of pose estimation learning.

**Definition of the loss function** After matching predictions and ground truth, each loss component is first weighted by the *assignment scores* from the matching algorithm, then normalized by the sum of these scores. This step stabilizes gradient scale by accounting for the actual number of objects and keypoints detected<sup>1</sup> :

$$\mathcal{L}_{\text{norm}} = \frac{\sum(\mathcal{L}_i \times \text{assigned score})}{\sum \text{assigned scores}}$$

The normalized losses are then combined with a weighted sum using fixed coefficients<sup>2</sup> :

$$\mathcal{L} = 1.0 \cdot \mathcal{L}_{\text{cls}} + 2.5 \cdot \mathcal{L}_{\text{iou}} + 0.5 \cdot \mathcal{L}_{\text{DFL}} + 1.0 \cdot \mathcal{L}_{\text{pose-cls}} + 1.0 \cdot \mathcal{L}_{\text{pose-reg}}$$

where :

- $\mathcal{L}_{\text{cls}}$  is the binary classification loss (presence/absence of an object),
- $\mathcal{L}_{\text{iou}}$  is the bounding box regression loss,
- $\mathcal{L}_{\text{DFL}}$  is the *Distribution Focal Loss* for precise edge localization,
- $\mathcal{L}_{\text{pose-cls}}$  is the keypoint visibility loss,
- $\mathcal{L}_{\text{pose-reg}}$  is the keypoint coordinate regression loss.

The detailed behavior of each component is presented in Appendix .3.

### 6.3.6 Fine-Tuning YOLO-NAS-POSE on LARD

I fine-tuned the YOLO-NAS-POSE-S model, which has 22.21M optimized parameters. The GPU used for the g5.xlarge instance is an **NVIDIA A10G Tensor Core** with 24 GB of memory.

Two training configurations were explored :

- a first scenario using COCO-POSE pre-trained weights, with a batch of 16 images on a single GPU ;
- a second scenario initializing the model with random weights, with a batch of 24 images (still on a single GPU).

In YOLO-NAS-POSE, **Average Precision (AP)** and **Average Recall (AR)** are computed only on keypoints. A predicted keypoint is considered a true positive if its **Object Keypoint Similarity (OKS)** with the reference keypoint satisfies :

$$\text{OKS}(\text{predicted keypoint}, \text{reference keypoint}) \geq \text{threshold} \quad (6.1)$$

Precision and recall for each keypoint are defined as :

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad R = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (6.2)$$

where :

- TP : number of correctly detected keypoints ( $\text{OKS} \geq \text{threshold}$ )

---

1. See implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L443](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L443)

2. See the source code implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L491](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L491)

- FP : number of incorrectly predicted keypoints ( $\text{OKS} < \text{threshold}$ )
- FN : number of missing keypoints

**Average Precision (AP)** and **Average Recall (AR)** are then computed as the mean over all keypoints or over different OKS thresholds :

$$\text{AP} = \frac{1}{N} \sum_{i=1}^N P(i), \quad \text{AR} = \frac{1}{M} \sum_{i=1}^M R(i) \quad (6.3)$$

where  $N$  and  $M$  respectively represent the number of points or thresholds considered.

TABLE 6.1 – Training performance comparison of YOLO-NAS-POSE-S with and without COCO pre-trained weights.

Setup	Epoch	Train Loss	Val Loss	AP	AP@0.50	AR
<b>COCO Weights</b>	50	2.2037	2.1310	0.2728	0.3052	0.5131
	98	1.7963	1.8772	0.2822	0.3168	0.5224
<b>No Weights</b>	50	2.7741	2.2378	0.2775	0.3085	0.5083
	100	2.1083	1.9026	0.2905	0.3217	0.5139
	198	1.6147	1.6510	0.2987	0.3355	0.5249

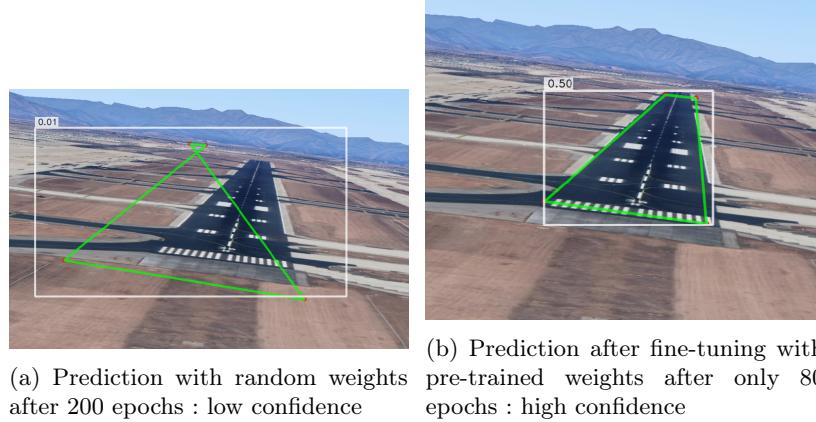


FIGURE 6.7 – Qualitative comparison of YOLO-NAS-POSE-S outputs on LARD.

TABLE 6.2 – Average confidence scores for keypoints and global detection on the 2,315-image test set.

Setup	Keypoint 1	Keypoint 2	Keypoint 3	Keypoint 4	All keypoints	Detection
COCO Weights	0.793	0.795	0.741	0.734	0.766	0.265
No Weights	0.300	0.300	0.271	0.269	0.285	0.077

These results highlight the difficulty of the problem : joint estimation of pose and keypoints is significantly more demanding than simple object detection. When trained from random weights, the model converges more slowly and predictions often remain unreliable (Figure 6.7a). Despite progressive improvement in global metrics ( $\text{AP} \approx 0.30$  and  $\text{AR} \approx 0.52$  after 200 epochs), average confidence scores remain very low, both for individual keypoints (0.27–0.30) and for global detection (0.077) (Table 6.2), limiting the practical usability of the results.

However, the *from-scratch* model is not incapable of learning : as early as 50 epochs, validation metrics ( $\text{AP} \approx 0.28$ ,  $\text{AR} \approx 0.51$ ) are comparable to those obtained with pre-trained weights. The network statistically detects the runway and its keypoints correctly, but score calibration remains insufficient, as indicated by low average confidence values (Table 6.2).

Conversely, using COCO-POSE pre-trained weights provides a decisive advantage : the model produces reliable detections with much higher confidence scores for keypoints (0.73–0.79) and for global detection (0.265) (Figure 6.7b

and Table 6.2). This better calibration of output probabilities translates into faster convergence and makes the model operational much earlier.

## 6.4 Toward an End-to-End Pipeline for Pose Estimation

### Introduction to the Pose Estimation Problem

Before presenting the two approaches—*partially end-to-end* and then *fully end-to-end*—it is essential to explain the value of 3D pose estimation in VLS. This information is central to enhancing system reliability.

Theoretically, pose estimation is formulated as an **inverse problem** : given the camera intrinsics (matrix  $A$ ) and correspondences between known 3D points (in a world frame) and their 2D projections in the image, we seek the camera rotation  $R$  and translation  $C$ .

This problem is well known as **PnP** (*Perspective-n-Point*) in computer vision. When  $n = 3$ , it is the **P3P** problem, which admits fast analytical solutions but potentially ambiguous ones (up to 4 solutions). Adding a fourth point generally lifts the ambiguity.

In our context, the 2D points are the *keypoints* detected by YOLO-NAS-POSE, while the associated 3D positions are known a priori thanks to the geometric modeling of the runway (at the time of data labeling).

### First approach : partially end-to-end pipeline

In a first approach, pose estimation is performed in the classical way via OpenCV's `solvePnP`. It takes 2D–3D correspondences and camera parameters as input and returns the camera position (vector  $C$ ) and orientation (matrix  $R$ ) in the world frame.

Estimation is carried out from the four runway corners detected in the image by YOLO-NAS-POSE.

### Coupling with YOLO-NAS-POSE

The pipeline operates sequentially :

1. YOLO-NAS-POSE detects the 2D keypoints (runway corners).
2. These points are then associated, in post-processing, with their corresponding 3D coordinates in the world frame.
3. The `solvePnP` algorithm estimates the camera pose.

However, this method has a major drawback : it does not allow the detector to benefit from pose information. Pose computation is entirely decoupled from training, preventing any joint improvement in detection and localization.

### Limitations of this approach

This partially end-to-end solution has several drawbacks :

- **Non-differentiability** : `solvePnP` is implemented only in C++<sup>3</sup>, which prevents incorporating the code into a framework like PyTorch.
- **No pose supervision** : pose quality is not accounted for in the loss function and thus not in network optimization.

While simple to implement, this first approach does not fully exploit the potential of integrated deep learning, particularly regarding geometric supervision.

This is why we propose a second approach, this time fully implemented in PyTorch, in which pose is integrated directly into learning via an appropriate loss function.

---

3. See C++ implementation : <https://github.com/opencv/opencv/blob/master/modules/calib3d/src/solvepnp.cpp>

## Second End-to-End Approach

### Motivation and Approach Overview

The goal of this approach is to enable YOLO-NAS-POSE to benefit from explicit supervision on the estimated 3D pose, not just on the 2D localization of keypoints. To this end, we designed a differentiable P3P implementation in PyTorch, based on the analytical formulation proposed by Kneip et al. [2011]. Development was carried out in collaboration with an intern under my supervisor's guidance. I also contributed to technical supervision, as the intern had no prior experience with PyTorch.

This implementation, presented in Appendix .4, directly extracts the camera rotation and translation from a triplet of 3D points and their 2D projections. Integrating this implementation makes it possible to use pose error during training, thereby reinforcing the model's geometric coherence. The associated code is available in various GitHub repositories<sup>4 5 6</sup>. It is important to stress that this part of the report is preliminary. Conducted under my supervisor's guidance, this work took place at the very end of my training, at a time when my autonomy had significantly increased.

#### Methodological Reminder : Differentiability and Optimization

In this section, we emphasize **differentiability**.

In deep learning, a function is **differentiable** if its slope (gradient) can be computed at every point. This means we can know in which direction to move the model parameters to reduce the loss.

*Concretely* : if we denote the loss by  $L(\theta)$  where  $\theta$  are the network parameters, we want the gradient

$$\nabla_{\theta} L(\theta)$$

to exist and be computable.

This is what allows the gradient descent algorithm (backpropagation) to work. Without derivatives, no learning.

**Why is this important here ?** Many modern losses (GIoU, CIoU, DFL, etc.) are designed to remain differentiable. Even if their formulas seem complex, they are written so that PyTorch can automatically compute gradients and use them for optimization.

Without this property, it would be impossible to use pose error as a learning signal in YOLO-NAS-POSE.

### Designing a Pose-Aware Loss

To improve robustness and accuracy of pose estimation in the pipeline, I designed a loss sensitive to scene geometry. Unlike the classical YOLO-NAS-POSE losses, which are limited to 2D keypoint accuracy (see 6.3.5), our loss explicitly integrates the camera position and orientation via solving the P3P problem.

The `loss_poseidon` function takes the camera intrinsic matrix, a set of 3D points in the world frame, their 2D projections from ground truth, and the network's 2D predictions. These are the same inputs needed to solve P3P, whose steps are detailed in Appendix .4.

Recall that solving P3P yields up to four potential solutions. In this implementation, solution selection was originally performed by a `hard-argmin`, i.e., directly picking the solution that minimizes the reprojection error. However, in

4. <https://github.com/alyaslt/poseidon> : my reference implementation of P3P in PyTorch and NumPy, and the differentiable loss in PyTorch.

5. <https://github.com/Pruneeuh/AutoRoot> : fully differentiable analytical solver library (cubic and quartic equations), developed by the intern.

6. <https://github.com/Pruneeuh/POSEIDON> : packaged version of the P3P implementation, also developed by the intern, which currently has a differentiability issue.

a differentiable context, a `hard-argmin` is not usable : it is discrete and its gradient is almost everywhere zero, preventing backpropagation. To remedy this, we use a *soft-argmin* that assigns a weight to each solution as a function of its reprojection error, maintaining a continuous gradient flow through selection.

Concretely, we use a differentiable approximation : *soft-argmin*. We associate with each candidate  $(\mathbf{R}_k, \mathbf{t}_k)$  a weight  $w_k$  from its reprojection error  $e_k$  :

$$w_k = \frac{\exp\left(-\frac{e_k}{\tau}\right)}{\sum_j \exp\left(-\frac{e_j}{\tau}\right)},$$

where  $\tau$  is a temperature parameter controlling selection hardness (small  $\tau$  approaches *hard-argmin*).

The final pose is then a weighted average of the candidate solutions :

$$\mathbf{R} = \sum_k w_k \mathbf{R}_k, \quad \mathbf{t} = \sum_k w_k \mathbf{t}_k.$$

In practice, this operation is implemented in PyTorch as :

```
w = torch.softmax(-reproj_errors / tau, dim=1) # (B, S)
```

To illustrate the *soft-argmin*, we simulated a case where four candidate solutions  $(\mathbf{R}, \mathbf{C})$  are generated by P3P. The matrices below present the solutions returned after solving P3P during the loss (truncated to 4 decimals) :

$$\text{Solutions} = \begin{bmatrix} 0.4027 & 0.8976 & -0.3707 & -0.2385 \\ 0.6145 & -0.4246 & -0.8723 & -0.2425 \\ 2.5759 & -0.1181 & 0.3189 & -0.9404 \end{bmatrix}, \begin{bmatrix} 0.4027 & 0.8976 & -0.3707 & -0.2385 \\ 0.6145 & -0.4246 & -0.8723 & -0.2425 \\ 2.5759 & -0.1181 & 0.3189 & -0.9404 \end{bmatrix},$$

$$\begin{bmatrix} 0.0012 & 1.0000 & -0.0005 & -0.0005 \\ 0.0021 & -0.0005 & -1.0000 & -0.0007 \\ 2.9989 & -0.0005 & 0.0007 & -1.0000 \end{bmatrix}, \begin{bmatrix} -1.1525 & 0.8259 & -0.5600 & -0.0642 \\ 0.8628 & 0.5559 & 0.8281 & -0.0724 \\ -1.1164 & 0.0937 & 0.0242 & 0.9953 \end{bmatrix}.$$

Applying the *softmax* to reprojection errors with a small  $\tau = 0.01$  yields :

$$w \approx [0.0000, 0.0000, 1.0000, 0.0000].$$

We observe that all probability mass concentrates on the third solution, which is practically a *hard-argmin* while preserving differentiability.

The final estimated solution is then given by the weighted-average rotation and camera center :

$$\mathbf{R} \approx \begin{bmatrix} 1.0000 & -0.0005 & -0.0005 \\ -0.0005 & -1.0000 & -0.0007 \\ -0.0005 & 0.0007 & -1.0000 \end{bmatrix}, \quad \mathbf{C} \approx (0.0012, 0.0021, 2.9989).$$

These results are consistent with the true pose defined in the simulation, where camera rotation and position were set by construction as :

$$\mathbf{R}_{\text{init}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad \mathbf{C}_{\text{init}} = \begin{bmatrix} 0 & 0 & 3 \end{bmatrix}.$$

Thus, the *soft-argmin* mechanism retrieves the most coherent pose while maintaining a differentiable computation chain.

Once the pose is estimated, 3D points are reprojected into the image via  $\mathbf{P} = \mathbf{A}[\mathbf{R} \mid \mathbf{t}]$ . The reprojection error is

computed as :

$$\mathcal{L}_{\text{reproj}} = \frac{1}{N} \sum_{i=1}^N \|\hat{\mathbf{p}}_i - \mathbf{p}_i\|_2$$

where  $\hat{\mathbf{p}}_i$  is the network prediction for point  $i$  and  $\mathbf{p}_i$  is the reprojection from the estimated pose.

### Conclusions and Integration Perspectives

This work is preliminary in nature. I am particularly proud of it, as it marks an important step in my growing autonomy during the final phase of my training.

The next challenge will be to integrate this *pose-aware loss* into YOLO-NAS-POSE training as soon as version 2 of the LARD dataset, currently in progress, becomes available. This new version will provide full 2D-3D correspondences for each labeled runway, a necessary condition for applying P3P. The POSEIDON library must also be finalized.

At this stage, to test the loss function, I had to simulate an artificial data stream by generating 2D-3D correspondences as well as noisy 2D predictions to reproduce realistic training conditions. Finally, I will also be responsible for knowledge transfer on this task to the next apprentice on the team to ensure development continuity.

# Conclusion and Professional Perspectives

In conclusion, I am deeply grateful to have been part of this team and to have contributed to the *Vision Landing System* project. My heartfelt thanks to Mélanie Ducoffe for her support and guidance throughout my apprenticeship. She was a true anchor for me and gave me confidence in my abilities.

This project was a real passion for me : I was able to improve my skills in PyTorch, my understanding of research papers, and my approach to mathematics. I managed to better grasp certain abstract concepts that used to intimidate me, and this year showed me that with time and hard work, I am capable.

I also made progress in public speaking : I am now more confident explaining my work, including complex concepts that I can make accessible. I have also significantly improved my English thanks to writing the paper and giving presentations (at ANITI in July, and soon in London, both entirely in English).

This work-study program (*alternance*) was perfectly aligned with my MIASHS master's program, which gave me the tools to understand many concepts in data science and artificial intelligence. In my case, this concerned deep learning in particular, where I already had foundations in NLP and Computer Vision, as well as solid command of the frameworks. We even had research-oriented courses with Mr. Pasquet, who encouraged us to question the state of the art. These last few months in industry extended and completed that training : I was able to deepen these aspects and begin to specialize. My supervisor challenged me in much the same way as Mr. Pasquet, which reinforced the consistency between my coursework and my experience in the company.

This first experience in an industrial research setting confirmed that this is a field I am passionate about. What I enjoy most is evaluating innovative solutions—and, above all, applying them in a real-world context. This experience has strengthened my desire to continue in research.

Of course, the field is competitive. I remain open to several paths : a PhD in computer vision, but also a position as a research engineer in a lab, ideally abroad. Immersion in English has given me the confidence to consider an international career.

That said, I have not yet chosen a precise specialization in deep learning, particularly between Computer Vision and Natural Language Processing. I am very interested in both, and I am still reflecting on which one best matches my aspirations and strengths. I will decide based on the opportunities that arise.

# Chapitre 7

# Appendix

## Sommaire

---

<b>.1</b>	<b>Training Details and Example Predictions for the Different Configurations</b>	<b>58</b>
.1.1	How to launch a training run ?	58
.1.2	Example of Metadata	59
.1.3	Example Predictions for the Configurations	60
<b>.2</b>	<b>C-mAP Implementation in Python</b>	<b>61</b>
<b>.3</b>	<b>Details on How the YOLO-NAS-POSE Losses Work</b>	<b>62</b>
<b>.4</b>	<b>Detailed Explanation of P3P following Kneip et al. [2011]</b>	<b>64</b>

---

## .1 Training Details and Example Predictions for the Different Configurations

### .1.1 How to launch a training run ?

Ultralytics lets you launch trainings using simple and flexible CLI (Command Line Interface) commands, with various arguments to customize the process. These commands adapt to multiple tasks and modes depending on the project's needs.

The main arguments we care about are :

- **TASK** (optional) : Sets the task to run, among : [detect, segment, classify, pose, obb].
- **MODE** (required) : Sets the execution mode, among : [train, val, predict, export, track, benchmark].

Configurable parameters include :

- **model** : Path to the model file to use for training. This can be a .pt checkpoint or a pretrained .yaml config file.
- **data** : Path to the dataset configuration file (e.g., coco8.yaml). This file contains the training/validation paths, class names, and number of classes.
- **epochs** : Total number of training epochs.
- **batch (int, default=16)** : Sets the training batch size. You can pass an integer value (e.g., **batch=16**).
- **imgsz (int or list, default=640)** : Target image size used for training. All images are resized to this before being fed to the model.
- **save (bool, default=True)** : Saves checkpoints during training as well as final model weights.

- **cache (bool, default=False)** : Enables dataset image caching either in RAM (`True/ram`), on disk (`disk`), or disabled (`False`). This can speed up training by reducing I/O, at the cost of higher memory use.
- **device (int, str or list, default=None)** : Specifies the compute device(s) for training.
- **project (str, default=None)** : Name of the directory where training results are stored. Useful to keep experiments organized.
- **name (str, default=None)** : A specific name for the training run. A subdirectory with this name will be created inside the project folder.

A typical training command may look like :

```
1 python tools/train.py --task detect --mode train --model yolov5s.yaml --data
coco8.yaml --epochs 50 --batch 16 --imgsz 640 --device 0 --project runs/
train --name yolov5_experiment
```

## 1.2 Example of Metadata

Below is one line from the `LARD_train.csv` file, formatted as a table for clarity :

Column	Value
<code>image</code>	<code>LARD_train_BIRK_LFST/images/BIRK_01_500_000.jpeg</code>
<code>height</code>	2648
<code>width</code>	2448
<code>type</code>	<code>earth_studio</code>
<code>original_dataset</code>	<code>LARD_train_BIRK_LFST</code>
<code>scenario</code>	<code>BIRK_01_500</code>
<code>airport</code>	<code>BIRK</code>
<code>runway</code>	1
<code>time_to_landing</code>	
<code>weather</code>	
<code>night</code>	
<code>time</code>	01/08/2020 15:22
<code>slant_distance</code>	4.15
<code>along_track_distance</code>	4.14
<code>height_above_runway</code>	1340.09
<code>lateral_path_angle</code>	-1.7
<code>vertical_path_angle</code>	2.93
<code>yaw</code>	-8.292996454006046
<code>pitch</code>	86.25697870547582
<code>roll</code>	-10.94856108654156
<code>watermark_height</code>	300.0
<code>x_A, , [...], y_D</code>	1323, 1271, 1339, 1274, 1293, 1295, 1312, 1298

TABLE 1 – Example of a metadata row.

### .1.3 Example Predictions for the Configurations

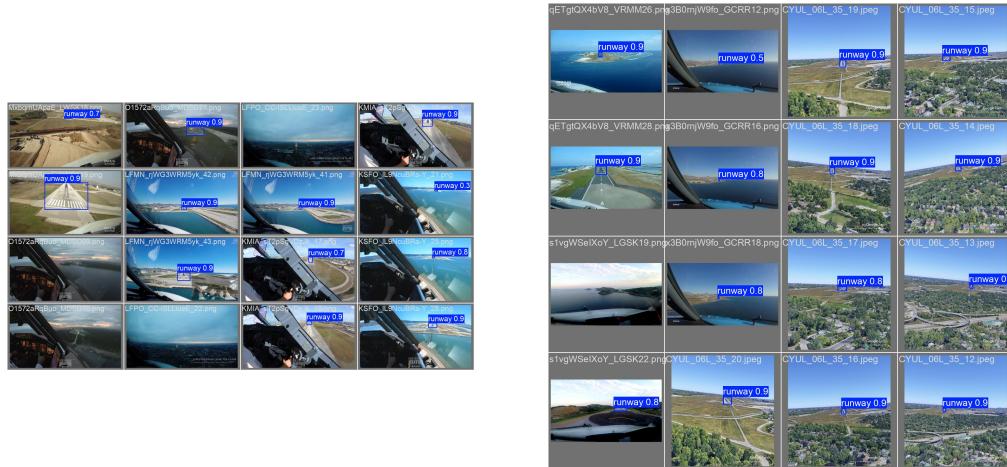


FIGURE 1 – Prediction Example for V5 Nano Pretrained during Trial 1.

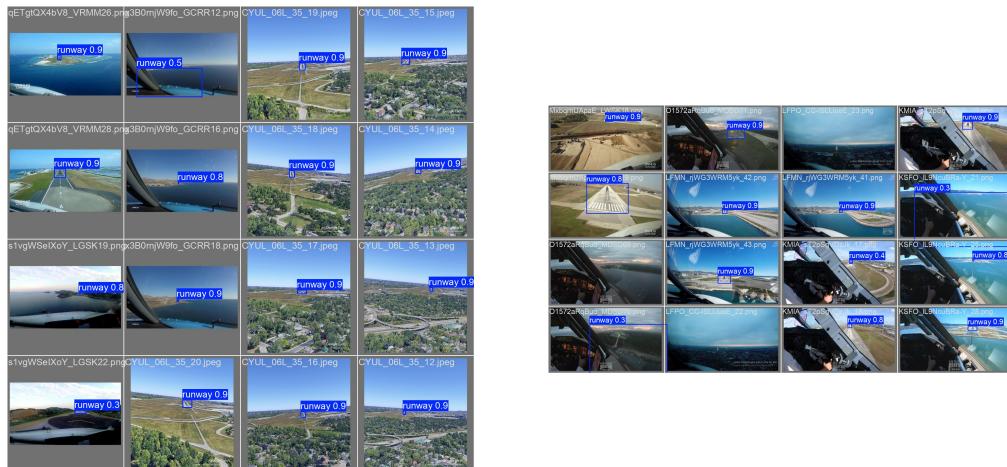


FIGURE 2 – Prediction Example for V5 Small Pretrained during Trial 1.



FIGURE 3 – Prediction Example for V5 Nano Random during Trial 2.



FIGURE 4 – Prediction Example for V5 Small Random during Trial 2.

## .2 C-mAP Implementation in Python

The function below computes *C-mAP* from a CSV file that contains, for each prediction, the score, IoU, and IoA :

```

1 def CmAP(filename, iou_threshold:float, ioa_threshold:float=1, iou_name="iou_pred_gt", ioa_name=""
2     ioa_pred_gt", verbose=0)->float:
3     # compute mAP from the csv files computed for the experiments of the paper
4
5     data = pd.read_csv(filename)
6     iou = data[iou_name].values
7     ioa = data[ioa_name].values
8     score = np.asarray([float(e.split(',')[1].split(']')[0]) for e in data['score'].values])
9     index_sort = np.argsort(score)
10    score_ = score[index_sort]
11    iou_ = iou[index_sort]
12    ioa_ = ioa[index_sort]
13
14    TP, FP, FN = c_map_order(iou= iou_, ioa= ioa, iou_threshold=iou_threshold, ioa_threshold=
15        ioa_threshold)
16
17    if verbose:
18        print('TP=', TP)
19        print('FP=', FP)
20        print('FN=', FN)
21
22    # compute precision
23    precision:ArrayLike = compute_precision(TP, FP)
24    precision = precision[1:]
25    if verbose:
26        print('precision=', precision)
27
28    # compute recall
29    recall:ArrayLike = compute_recall(TP, FN)
30    recall = recall[1:]
31    if verbose:
32        print('recall=', recall)
33
34    # compute over approximation of the plot precision/recall
35    recall, precision = over_approximate(recall, precision)
36    if verbose:
37        print('over_approx', recall, precision)
38
39
40    return coco_sampling(recall, precision)

```

### .3 Details on How the YOLO-NAS-POSE Losses Work

**Classification loss  $\mathcal{L}_{\text{cls}}$**  In YOLO-NAS-Pose, the classification task predicts, for each candidate box, whether it contains a person or not. Since the model detects a single class (“human”), this is a **binary** classification problem : predict the probability that the box contains a person.

Let  $z \in \mathbb{R}$  be the logit output by the model, and  $p = \sigma(z) \in (0, 1)$  its probability after a sigmoid :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The ground-truth label  $y \in \{0, 1\}$  is :

- $y = 1$  if the box is assigned to a person (ground truth),
- $y = 0$  if the box corresponds to background (negative).

The goal is  $p \approx 1$  when  $y = 1$ , and  $p \approx 0$  when  $y = 0$ . The error is measured with a loss that is always a **non-negative real** (never simply 0 or 1).

Two functions<sup>1 2</sup> are used depending on configuration :

- **Binary Cross-Entropy with Logits (BCEWithLogits)** :

$$\mathcal{L}_{\text{BCE}}(z, y) = -[y \cdot \log(\sigma(z)) + (1 - y) \cdot \log(1 - \sigma(z))]$$

This returns a positive real value that increases as  $p$  deviates from  $y$ .

- **Focal Loss** Lin et al. [2018], designed to better handle class imbalance (background vs. person) :

$$\mathcal{L}_{\text{focal}}(p, y) = \alpha_t (1 - p_t)^\gamma \cdot \mathcal{L}_{\text{BCE}}(p, y)$$

with :

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases}, \quad \alpha_t = \begin{cases} \alpha & \text{if } y = 1 \\ 1 - \alpha & \text{if } y = 0 \end{cases}$$

Here,  $\gamma$  focuses the loss on hard examples, and  $\alpha$  balances the classes.

In both cases, the loss output is strictly positive, close to zero when the prediction is correct (e.g.,  $p \approx 1$  for  $y = 1$ ), and large when it is incorrect (e.g.,  $p \approx 0$  for  $y = 1$ ).

**Bounding-box regression loss  $\mathcal{L}_{\text{iou}}$**  This component measures localization error between predicted boxes and ground-truth boxes. Instead of  $\ell_1/\ell_2$  regression losses, IoU-based variants are used to incorporate geometric cues. Two options are GIoU and CIoU.

- **GIoU (Generalized IoU)** Rezatofighi et al. [2019] extends IoU by penalizing non-overlapping cases. It is defined<sup>3</sup> as :

$$\mathcal{L}_{\text{GIoU}} = 1 - \left( \text{IoU} - \frac{|\mathcal{C} \setminus (\mathcal{A} \cup \mathcal{B})|}{|\mathcal{C}|} \right)$$

where :

- $\mathcal{A}$  is the area of the predicted box,
- $\mathcal{B}$  is the area of the ground-truth box,
- $\mathcal{C}$  is the area of the smallest enclosing box.

1. See the implementation in the source code : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L664](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L664)

2. See the implementation in the source code : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L664](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L664)

3. See implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/pyyolo\\_loss.py#L609](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/pyyolo_loss.py#L609)

The leading  $1 - \cdot$  turns the score (in  $[-1, 1]$ ) into a positive loss to minimize.

- **CIoU (Complete IoU)** Zheng et al. [2019] further refines geometry by combining<sup>4</sup> :
  1. the direct IoU loss,
  2. center distance,
  3. aspect-ratio divergence.

The full formula is :

$$\mathcal{L}_{\text{CIoU}} = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}^g)}{c^2} + \alpha v$$

where :

- $\rho^2(\mathbf{b}, \mathbf{b}^g)$  is the squared distance between centers,
- $c^2$  is the diagonal of the minimal enclosing box,
- $v = \frac{4}{\pi^2} (\arctan(\frac{w^g}{h^g}) - \arctan(\frac{w}{h}))^2$ ,
- $\alpha = \frac{v}{(1-\text{IoU})+v}$  is a dynamic weight (non-differentiable) for  $v$ .

CIoU stabilizes training by considering overlap, relative position, and shape.

**Distribution Focal Loss Li et al. [2020]**  $\mathcal{L}_{\text{DFL}}$  *Distribution Focal Loss* (DFL) improves the precision of continuous-value regression such as keypoint coordinates. Instead of direct regression (L1/MSE), DFL reformulates it as classification over  $C$  discrete bins.

In the original paper Li et al. [2020],  $C = 256$ , balancing accuracy and efficiency. Note that the exact  $C$  used by Deci-AI's implementation is not explicitly documented in public code.<sup>5</sup>

Rather than predicting a real coordinate  $t \in \mathbb{R}$ , the model predicts a probability distribution  $p = [p_0, \dots, p_{C-1}]$  over  $\{0, \dots, C-1\}$ . The real value is bracketed by  $t_L = \lfloor t \rfloor$  and  $t_R = t_L + 1$ , and the loss applies a weighted interpolation of two cross-entropies :

$$\mathcal{L}_{\text{DFL}}(p, t) = (1 - w) \cdot \text{CE}(p, t_L) + w \cdot \text{CE}(p, t_R), \quad \text{with } w = t - t_L.$$

This discretization offers classification stability while retaining continuous precision, which explains its effectiveness in modern architectures like YOLO-NAS.

**Pose estimation loss** The pose-estimation task has two parts :

**Coordinate regression ( $\mathcal{L}_{\text{pose-reg}}$ )** The keypoint regression loss<sup>6</sup> is computed per keypoint  $i$  as :

$$e_i = \frac{\|\hat{\mathbf{x}}_i - \mathbf{x}_i\|^2}{(2\sigma_i)^2 \cdot s^2 \cdot 2}$$

where

- $\hat{\mathbf{x}}_i$  are predicted coordinates,
- $\mathbf{x}_i$  are ground-truth coordinates,
- $\sigma_i$  is per-joint tolerance (std),
- $s^2$  is the area of the bounding box,
- $v_i$  is the keypoint visibility (1 if visible, else 0).

4. See implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/functional.py#L82](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/functional.py#L82)

5. See implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L500](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L500)

6. Reference implementation : [https://github.com/Deci-AI/super-gradients/blob/master/src/super\\_gradients/training/losses/yolo\\_nas\\_pose\\_loss.py#L546](https://github.com/Deci-AI/super-gradients/blob/master/src/super_gradients/training/losses/yolo_nas_pose_loss.py#L546)

The unreduced loss for each point is :

$$\ell_i = 1 - \exp(-e_i)$$

The final loss averages over visible points :

$$\mathcal{L}_{\text{pose-reg}} = \frac{\sum_{i=1}^N \ell_i \cdot \mathbb{1}_{v_i > 0}}{\sum_{i=1}^N \mathbb{1}_{v_i > 0} + \varepsilon}$$

with  $\mathbb{1}_{\{v_i > 0\}}$  the indicator of visibility.

**Visibility classification ( $\mathcal{L}_{\text{pose-cls}}$ )** : For each keypoint  $i$ , the model predicts a visibility  $\hat{v}_i \in [0, 1]$  compared to the ground truth  $v_i \in \{0, 1\}$ . This is a binary classification problem per keypoint, defined by either :

— **Binary Cross Entropy (BCE)** :

$$\mathcal{L}_{\text{pose-cls}} = \text{BCEWithLogits}(\hat{v}_i, v_i)$$

— or **Focal Loss**, as defined previously, to better handle visible/non-visible imbalance.

## .4 Detailed Explanation of P3P following Kneip et al. [2011]

The **PnP** (Perspective- $n$ -Point) problem estimates camera pose from correspondences between 3D world points and their 2D projections. This requires knowing the camera intrinsics matrix  $\mathbf{A} \in \mathbb{R}^{3 \times 3}$ , and at least  $n \geq 3$  2D–3D correspondences.

In the VLS project, the LARD dataset provides precise 2D runway annotations (bounding boxes and the four runway corner pixels). However, this version lacks world-aligned 3D correspondences. LARD V2 will provide them ; then the four anchors will serve as usable 2D/3D correspondences to estimate camera pose. Meanwhile, we use three artificially generated 2D/3D pairs to test our **P3P** implementation in PyTorch.

**P3P** (Perspective-Three-Point) is a fundamental case : estimate camera *pose*, i.e., camera position  $\mathbf{C} \in \mathbb{R}^3$  and orientation  $\mathbf{R} \in \text{SO}(3)$  from geometric measurements.

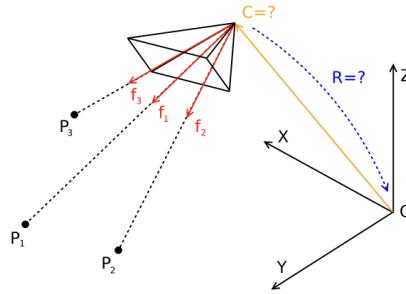


FIGURE 5 – Illustration of pose estimation from Kneip et al. [2011] (P3P).

Inputs :

- three known world points  $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$  (an optional  $\mathbf{P}_4$  may be used for disambiguation) ;
- three image points  $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ , obtained by perspective projection of the world points via the camera model.

From these 2D points and the intrinsics, we compute three unit direction vectors in the camera frame,  $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$ , representing the incident ray directions from the camera center to the world points.

The problem admits up to four real solutions. We follow :

*“A Novel Parametrization of the Perspective-Three-Point Problem for a Direct Computation of Absolute Camera Position and Orientation” Laurent Kneip, Davide Scaramuzza, Roland Siegwart, CVPR.*

This gives a direct, analytical solution from  $\mathbf{f}_i$  and  $\mathbf{P}_i$ . A fourth point can be used to evaluate reprojection error and pick the most plausible solution.

In practice, 2D image points  $\mathbf{p}_i$  are extracted via feature detectors (SIFT, ORB, etc.), then directions are obtained by backprojection with the intrinsics :

$$\mathbf{f}_i = \frac{A^{-1}\tilde{\mathbf{p}}_i}{\|A^{-1}\tilde{\mathbf{p}}_i\|},$$

where  $\tilde{\mathbf{p}}_i$  is  $\mathbf{p}_i$  in homogeneous coordinates.

To validate our P3P implementation in a controlled setting, we generate synthetic but coherent data :

$$\mathbf{P}_1 = \begin{bmatrix} -1.0727 \\ -1.6850 \\ -0.1344 \end{bmatrix}, \quad \mathbf{P}_2 = \begin{bmatrix} -0.3425 \\ -0.4382 \\ -0.9213 \end{bmatrix}, \quad \mathbf{P}_3 = \begin{bmatrix} -1.2048 \\ -1.0631 \\ -0.9224 \end{bmatrix},$$

and a fourth point for disambiguation :

$$\mathbf{P}_4 = \begin{bmatrix} 1.6402 \\ 1.2391 \\ -1.8479 \end{bmatrix}.$$

We specify a camera pose :

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 & 0 & 6 \end{bmatrix}^\top,$$

and intrinsics :

$$\mathbf{A} = \begin{bmatrix} 800 & 0 & 320 \\ 0 & 800 & 240 \\ 0 & 0 & 1 \end{bmatrix}.$$

World points are projected by :

$$\mathbf{p}_i = \mathbf{A} \cdot [\mathbf{R} | -\mathbf{RC}] \cdot \tilde{\mathbf{P}}_i,$$

then dehomogenized to get the observed 2D pixels. These perfect (noise-free) correspondences form an ideal base to assess P3P accuracy.

**Important note :** in real use, the pose ( $\mathbf{R}$ ,  $\mathbf{C}$ ) is unknown and must be estimated from measurements. 2D–3D correspondences come from detection and matching (e.g., SIFT, ORB, COLMAP, etc.). Building these correspondences is a necessary preprocessing step before P3P.

From the projected 2D points we build unit direction vectors in the camera frame.

The obtained 2D pixels are :

$$\mathbf{p}_1 = \begin{bmatrix} 455.7761 \\ 137.0256 \end{bmatrix}, \quad \mathbf{p}_2 = \begin{bmatrix} 187.9764 \\ 145.0786 \end{bmatrix}, \quad \mathbf{p}_3 = \begin{bmatrix} 123.5423 \\ 184.6140 \end{bmatrix}$$

For each  $\mathbf{p}_i$ , we :

1. Make it homogeneous :  $\tilde{\mathbf{p}}_i = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$ .

2. Backproject with  $A^{-1}$  :

$$\mathbf{f}'_i = \mathbf{A}^{-1} \cdot \tilde{\mathbf{p}}_i$$

3. Normalize to unit length :

$$\mathbf{f}_i = \frac{\mathbf{f}'_i}{\|\mathbf{f}'_i\|}$$

These  $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$  are in the camera frame—ray directions from the optical center to  $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$ .

*“We assume that the unitary vectors  $\mathbf{f}_1, \mathbf{f}_2$ , and  $\mathbf{f}_3$ —pointing toward the three considered feature points from the camera frame—are given.”* — Kneip et al. [2011] .

This preprocessing turns raw 2D points into geometric directions usable by the analytical model.

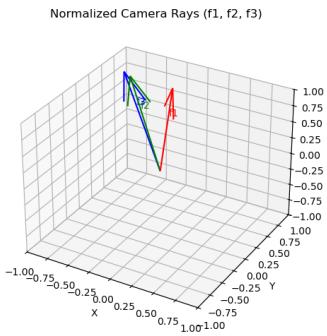


FIGURE 6 – Our feature vectors.

In summary, our P3P implementation takes three world points  $\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3$  and unit direction vectors  $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$  in the camera frame. These are obtained from the 2D points by backprojection with the intrinsics.

First, we ensure the three 3D points are not collinear—necessary so P3P is well-posed with finitely many solutions.

Next, we build an orthonormal basis in the camera frame from  $\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3$  to simplify later trigonometry. The image-frame basis 7 is :

$$\mathbf{e}_1 = \mathbf{f}_1, \quad \mathbf{e}_3 = \frac{\mathbf{f}_1 \times \mathbf{f}_2}{\|\mathbf{f}_1 \times \mathbf{f}_2\|}, \quad \mathbf{e}_2 = \mathbf{e}_3 \times \mathbf{e}_1.$$

The change-of-basis matrix  $\mathbf{T}$  is :

$$\mathbf{T} = \begin{bmatrix} \mathbf{e}_1^\top \\ \mathbf{e}_2^\top \\ \mathbf{e}_3^\top \end{bmatrix}, \quad \mathbf{f}_3^\tau = \mathbf{T}\mathbf{f}_3.$$

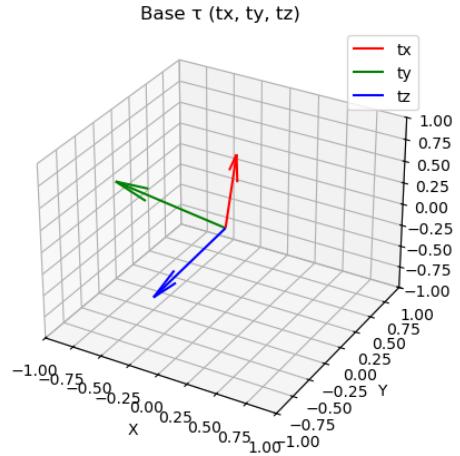


FIGURE 7 – Image basis associated with the direction vectors.

In parallel, we build an orthonormal basis attached to the world points,  $\eta = (\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z) :$

$$\mathbf{n}_x = \frac{\mathbf{P}_2 - \mathbf{P}_1}{\|\mathbf{P}_2 - \mathbf{P}_1\|}, \quad \mathbf{n}_z = \frac{\mathbf{n}_x \times (\mathbf{P}_3 - \mathbf{P}_1)}{\|\mathbf{n}_x \times (\mathbf{P}_3 - \mathbf{P}_1)\|}, \quad \mathbf{n}_y = \mathbf{n}_z \times \mathbf{n}_x.$$

Its change-of-basis matrix is :

$$\mathbf{N} = \begin{bmatrix} \mathbf{n}_x^\top \\ \mathbf{n}_y^\top \\ \mathbf{n}_z^\top \end{bmatrix}, \quad \mathbf{P}_3^\eta = \mathbf{N}(\mathbf{P}_3 - \mathbf{P}_1).$$

The paper then introduces intermediate variables :

- $\phi_1 = \frac{f_{3,x}^\tau}{f_{3,z}^\tau}$ ,  $\phi_2 = \frac{f_{3,y}^\tau}{f_{3,z}^\tau}$  from  $\mathbf{f}_3$  in the image basis ;
- $p_1, p_2$ , the first two components of  $\mathbf{P}_3^\eta$  ;
- $d_{12} = \|\mathbf{P}_2 - \mathbf{P}_1\|$  ;
- $\cos(\beta)$ , the cosine between  $\mathbf{f}_1$  and  $\mathbf{f}_2$  ;
- $b = \cot(\beta) = \frac{\cos(\beta)}{\sqrt{1-\cos^2(\beta)}}$ , with sign flip if  $\cos(\beta) < 0$ .

The problem is reformulated as a quartic :

$$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0,$$

whose coefficients depend on the variables above. Solving yields  $\theta$ -related values.

Analytical expressions for the coefficients are given in Kneip's paper.

For each real root  $\cos(\theta)$  (up to four), we reconstruct a valid camera pose :

1. **Compute**  $\sin(\theta)$  :

$$\sin(\theta) = \sqrt{1 - \cos^2(\theta)}.$$

2. **Compute**  $\alpha$  via its cotangent :

$$\cot(\alpha) = \frac{\left(\frac{\phi_1}{\phi_2}\right)p_1 + \cos(\theta)p_2 - d_{12}b}{\left(\frac{\phi_1}{\phi_2}\right)\cos(\theta)p_2 - p_1 + d_{12}}.$$

Then :

$$\sin(\alpha) = \sqrt{\frac{1}{\cot^2(\alpha) + 1}}, \quad \cos(\alpha) = \sqrt{1 - \sin^2(\alpha)}.$$

If  $\cot(\alpha) < 0$ , set  $\cos(\alpha) := -\cos(\alpha)$ .

### 3. Reconstruct camera center $\mathbf{C}_{\text{est}}$ :

$$\mathbf{C}_{\text{interm}} = d_{12} \cdot \begin{bmatrix} \cos(\alpha)(\sin(\alpha)b + \cos(\alpha)) \\ \sin(\alpha)\cos(\theta)(\sin(\alpha)b + \cos(\alpha)) \\ \sin(\alpha)\sin(\theta)(\sin(\alpha)b + \cos(\alpha)) \end{bmatrix}, \quad \mathbf{C}_{\text{est}} = \mathbf{P}_1 + \mathbf{N}^\top \cdot \mathbf{C}_{\text{interm}}.$$

### 4. Build rotation $\mathbf{R}_{\text{est}}$ with an intermediate :

$$\mathbf{Q} = \begin{bmatrix} -\cos(\alpha) & -\sin(\alpha)\cos(\theta) & -\sin(\alpha)\sin(\theta) \\ \sin(\alpha) & -\cos(\alpha)\cos(\theta) & -\cos(\alpha)\sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix}, \quad \mathbf{R}_{\text{est}} = \mathbf{N}^\top \cdot \mathbf{Q}^\top \cdot \mathbf{T}.$$

### 5. Store the solution :

e.g., as a  $(4, 3, 4)$  tensor across solutions, with the first column as  $\mathbf{C}_{\text{est}}$  and the next three as  $\mathbf{R}_{\text{est}}$ .

Repeat for each real root, then evaluate via reprojection to select the best.

A fourth point  $\mathbf{P}_4$  with its projection  $\mathbf{p}_4$  is used to disambiguate by comparing reprojection errors.

Given each candidate  $(\mathbf{R}_{\text{est}}, \mathbf{C}_{\text{est}})$ , we simulate the projection :

$$\mathbf{p}_i^{\text{est}} \sim \mathbf{A} \cdot \mathbf{R}_{\text{est}} \cdot (\mathbf{P}_i - \mathbf{C}_{\text{est}})$$

then dehomogenize to pixel coordinates.

We compare each  $\mathbf{p}_i^{\text{est}}$  to its observed  $\mathbf{p}_i^{\text{obs}}$  using reprojection error :

$$e_i = \|\mathbf{p}_i^{\text{est}} - \mathbf{p}_i^{\text{obs}}\|$$

and average over points :

$$\text{mean error} = \frac{1}{n} \sum_{i=1}^n e_i$$

We keep the solution minimizing this error and reject the others. This reprojection-based validation is crucial because P3P can yield several mathematically valid solutions ; comparing 2D projections to observations identifies the one that best matches reality.

# Bibliographie

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna : A next-generation hyperparameter optimization framework, 2019. URL <https://arxiv.org/abs/1907.10902>.

Léo Andéol, Thomas Fel, Florence De Grancey, and Luca Mossina. Confident object detection via conformal prediction and conformal risk control : an application to railway signaling. In *Conformal and Probabilistic Prediction with Applications*, pages 36–55. PMLR, 2023.

Florence de Grancey, Jean-Luc Adam, Lucian Alecu, Sébastien Gerchinovitz, Franck Mamalet, and David Vigouroux. Object detection with probabilistic guarantees. In *Fifth International Workshop on Artificial Intelligence Safety Engineering (WAISE 2022)*, 2022.

Mélanie Ducoffe, Maxime Carrere, Léo Féliers, Adrien Gauffriau, Vincent Mussot, Claire Pagetti, and Thierry Sammour. Lard – landing approach runway detection – dataset for vision based landing, 2023. URL <https://arxiv.org/abs/2304.09938>.

Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features, 2019. URL <https://arxiv.org/abs/1905.02175>.

Glenn Jocher. Ultralytics yolov5, 2020. URL <https://github.com/ultralytics/yolov5>.

Rahima Khanam and Muhammad Hussain. What is yolov5 : A deep look into the internal features of the popular object detector. *arXiv preprint*, 2407(20892v1), 2024. URL <https://arxiv.org/html/2407.20892v1>.

Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation. In *CVPR 2011*, pages 2969–2976, 2011. doi : 10.1109/CVPR.2011.5995464.

Harold W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1–2) : 83–97, March 1955. doi : 10.1002/nav.3800020109.

Chuyi Li, Lulu Li, Yifei Geng, Hongliang Jiang, Meng Cheng, Bo Zhang, Zaidan Ke, Xiaoming Xu, and Xiangxiang Chu. Yolov6 v3.0 : A full-scale reloading, 2023. URL <https://arxiv.org/abs/2301.05586>.

Xiang Li, Wenhui Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss : Learning qualified and distributed bounding boxes for dense object detection, 2020. URL <https://arxiv.org/abs/2006.04388>.

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. URL <https://arxiv.org/abs/1708.02002>.

Mouhcine Mendil, Luca Mossina, and David Vigouroux. Puncc : a python library for predictive uncertainty calibration and conformalization. In *Conformal and Probabilistic Prediction with Applications*, pages 582–601. PMLR, 2023.

Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. In *18th international conference on pattern recognition (ICPR'06)*, volume 3, pages 850–855. IEEE, 2006.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once : Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016. doi : 10.1109/CVPR.2016.91.

Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union : A metric and a loss for bounding box regression, 2019. URL <https://arxiv.org/abs/1902.09630>.

Rejin Varghese and Sambath M. Yolov8 : A novel object detection algorithm with enhanced performance and robustness. In *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, pages 1–6, 2024. doi : 10.1109/ADICS58448.2024.10533619.

Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss : Faster and better learning for bounding box regression, 2019. URL <https://arxiv.org/abs/1911.08287>.