Department of Electrical engineering -
Computer Vision Laboratory (CVL)

Internship report

Designing Computer Vision Pipelines: Video Inpaiting and 3D Reconstruction Pipelines

Swedish supervisor:
Michael FELSBERG
Professor, Head of Division

French Supervisor:
Christophe DE VAULX
Associate Professor

Student:
Mathis PICASSE

Location: Linköping, Sweden
Internship period: March 2025 – August 2025

Acknowledgement

I would like to thank **Michael Felsberg** for giving me the opportunity to be part of such an active and stimulating research environment at the Computer Vision Laboratory. It allowed me to meet many brilliant researchers and PhD students, and to better understand the challenges and rigor of academic research. The intellectually fulfilling atmosphere, combined with insightful discussions with lab members, greatly enriched my knowledge in the field of Artificial Intelligence. Being surrounded by such expertise was also a strong source of motivation to improve my own skills - a healthy reminder of the so-called impostor syndrome.

I am particularly grateful to **Mårten Wadenbäck**, who closely supervised me during the first month of the internship and was always available to provide support and guidance. His help was greatly appreciated, even though I later transitioned to a different project.

I would also like to thank **Pi-Erik Forssén** for giving me the opportunity to attend the 3D Computer Vision course. It was a demanding but highly rewarding learning experience.

Finally, I would like to thank **James Adam**, **Maëva Caut**, and **Carl Hoffstedt** for our collaboration during the group project in the 3D Computer Vision course. I am also thankful to **Johannes Hägerlind** for recommending the video inpainting project, which turned out to be a great learning experience.

Abstract

This six-month internship was conducted at the Computer Vision Laboratory (CVL) of Linköping University, Sweden, a renowned research center specializing in advanced perception and mathematical modeling. The main objective was to develop a fully automated video inpainting pipeline, addressing challenges in object detection, segmentation, and video reconstruction.

Working predominantly autonomously, I began by analyzing existing approaches and identifying suitable methods. The project involved designing and implementing a video inpainting pipeline using Python, along with cutting-edge models like YOLO for object detection and SAM for segmentation. The iterative development process resulted in a functional prototype demonstrated through a practical video inpainting demo.

Alongside this project, I attended an intensive 3D Computer Vision course provided by the lab, which enhanced my theoretical understanding and technical skills. The combination of coursework and project work allowed me to deepen my expertise in object detection, segmentation, video inpainting, and 3D reconstruction.

Keywords: Video Inpainting, Object Detection, Segmentation, Structure from Motion, Pipeline

Résumé

Ce stage de six mois s'est déroulé au sein du Laboratoire de Vision par Ordinateur (CVL) de l'Université de Linköping, en Suède, un centre de recherche reconnu spécialisé dans la perception avancée et la modélisation mathématique. L'objectif principal était de développer une pipeline entièrement automatisée pour l'inpainting vidéo, répondant aux défis de la détection d'objets, de la segmentation et de la reconstruction vidéo.

Travaillant principalement en autonomie, j'ai commencé par analyser les approches existantes et identifier les méthodes adaptées. Le projet a consisté à concevoir et implémenter une pipeline utilisant Python, ainsi que des modèles de pointe tels que YOLO pour la détection d'objets et SAM pour la segmentation. Le processus de développement itératif a abouti à un prototype fonctionnel, démontré par une application pratique d'inpainting vidéo.

En parallèle, j'ai suivi un cours intensif sur la vision par ordinateur 3D dispensé par le laboratoire, qui a renforcé mes connaissances théoriques et compétences techniques. La combinaison de ce cours et du projet m'a permis d'approfondir mon expertise en détection d'objets, segmentation, inpainting vidéo et reconstruction 3D.

Mots clefs: Inpainting vidéo, Détection d'objets, Segmentation, Structure à partir du Mouvement, Pipeline

Introduction

As a fourth-year engineering student, I undertook a six-month internship abroad to deepen my expertise in the rapidly evolving field of Computer Vision. I had the opportunity to complete this internship at the Computer Vision Laboratory (CVL) of Linköping University (LiU) in Sweden, a highly reputed international research center specializing in high-level perception problems and mathematical modeling.

Computer Vision addresses complex technical challenges with significant scientific and economic implications, both for society and the companies leveraging AI technologies. At the CVL, most researchers and PhD students focus on difficult tasks such as network infrastructure optimization. Initially, my contribution was limited by my relatively basic understanding of Computer Vision, as my prior knowledge was mainly restricted to Convolutional Neural Networks (CNNs), which form the backbone of many Computer Vision architectures.

During the first few weeks, I concentrated on identifying a suitable project aligned with my interest in object detection and segmentation, aiming to enhance my skills. This led me to develop a fully automated video inpainting pipeline, a project proposed by a PhD student, which allowed me to apply and deepen my understanding practically.

In parallel, I attended an intensive 3D Computer Vision course taught by a lab researcher, which was demanding but highly enriching. Due to the course's intensity and time requirements, I postponed some progress on the video inpainting project.

This report mainly details the development of the video inpainting pipeline and, in its final section, presents quickly the 3D Computer Vision course and its practical group project. I omit some of the more theoretical mathematical background and algorithms, as both the group project and the course topic could have constituted full internship projects on their own.

Contents

List of Figures

List of Tables

1. Discovering Linköping University

1.1 Dive into Linköping University

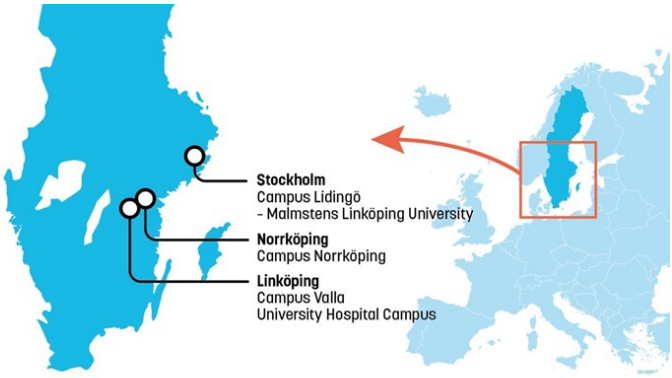


Figure 1.1: Map of Linköping University's campuses across Sweden

Linköping University (LiU) is a public research university located in Linköping, Sweden. Established in 1975, it has rapidly grown into one of the country's most prestigious institutions, renowned for its strong interdisciplinary approach and close ties to industry and the public sector.

As illustrated in Figure 1.1, LiU comprises four main campuses across Sweden:

- **Linköping**: Campus Valla and the University Hospital Campus
- **Norrköping**: Campus Norrköping
- **Stockholm**: Campus Lidingö

Today, the university community consists of approximately 50,000 individuals, including students, faculty, and staff. LiU consistently ranks among the top 2% of universities worldwide, according to the three major global university rankings:

- **201–250**: Times Higher Education World University Rankings
- **304**: QS World University Rankings
- **301**: Academic Ranking of World Universities (Shanghai Ranking)

LiU is internationally recognized for its groundbreaking research and innovation. It fosters collaboration not only across academic disciplines but also between academia, industry, civil society, and the public sector. This model of cooperation led to Linköping being awarded the title of *European Capital of Innovation* by the European Commission in 2023, the first Swedish city to receive this distinction. The award honors cities that implement innovative solutions to address social and environmental challenges.

One notable example of this commitment to sustainability is the deployment of the world's first biogas-powered buses in the streets of Linköping in the early 1990s. Biogas, produced locally from organic waste, contributes to cleaner air and water, renewable energy supply, and reduced greenhouse gas emissions.

1.2 The Computer Vision Laboratory (CVL)

Figure 1.2: Campus Valla, Linköping.

During my internship, I was based on Campus Valla, at the Computer Vision Laboratory (CVL), located in the B-building (see Fig.1.2). The CVL, led by Professor Michael Felsberg, is part of the Department of Electrical Engineering at Linköping University.

Research at CVL is theory-driven, with a strong foundation in machine learning, signal processing, and applied mathematics. These methods are applied to a wide range of domains including autonomous driving, sustainable agriculture and forestry, greenhouse gas monitoring, and wildlife tracking.

CVL's main research themes include:

- Continuous-time modeling of 3D motion
- Pose and 3D structure estimation
- Geometric deep learning
- Human and animal motion analysis
- Medical imaging and diagnostics
- Reinforcement learning
- Video and semantic segmentation

CVL is actively involved in several European research initiatives and maintains collaborations with leading institutions and programs such as WASP (Wallenberg AI, Autonomous Systems and Software Program) and ELLIIT (Excellence Center in ICT and Embedded Systems).

Here are some of the lab's most notable achievements:

- Widely cited publications in top-tier conferences:
  - Computer Vision and Pattern Recognition (CVPR)
  - European Conference on Computer Vision (ECCV)
  - Neural Information Processing Systems (NeurIPS)
- Real-world applications used by automotive and medical industries
- Patents and technology transfers in object detection and 3D reconstruction
- Contributions to open-source libraries and annotated datasets

1.3 Actions for Sustainable Development

Linköping University (LiU) has implemented a series of concrete actions to reduce its environmental impact:

- A structured environmental management system inspired by ISO 14001 guides decision-making and continuous improvement.
- Mandatory online sustainability training is provided to all staff to increase awareness and competence.
- Carbon emissions from air travel are tracked, with a target to reduce them by 30% (not yet reached), and flights are compensated through climate offsetting.
- Campus energy use has been reduced by 20% between 2019 and 2022, exceeding the 10% target.
- Green procurement practices are in place, with environmental criteria applied to energy- and climate-intensive goods and services.
- Reuse initiatives such as *Återbruket* promote the circular use of furniture and lab equipment.
- Hazardous chemical handling is improved through cross-departmental collaboration and enhanced safety routines.
- LiU is a signatory of the national Climate Framework, working alongside other Swedish universities to reduce emissions in travel, procurement, and energy use.

These actions reflect LiU's ambition to align with the UN's Agenda 2030 and integrate sustainability into its operations at every level.

2. Video Inpainting Pipeline

Upon my arrival at the Computer Vision Laboratory (CVL), my initial weeks were dedicated to a dual objective: identifying a suitable internship topic while simultaneously immersing myself in the lab's research environment. This period was marked by an immersion into a rich landscape of advanced concepts and specialized terminology. Terms such as object detection, pose estimation, and image segmentation were frequently encountered. Given that my foundational knowledge in Computer Vision primarily revolved around the mechanisms of Convolutional Neural Networks (CNNs), it became clear that a dedicated effort was needed to bridge this knowledge gap and master these specific applications.

An initial project involving the calibration of a robotic arm was proposed. However, driven by my strong interest in core computer vision algorithms, I took the initiative to propose a new direction centered on video analysis. This proposal was well-received, and I was encouraged to discuss it with a PhD student working on a related subject. Our conversations were pivotal; they led to the definition of a concrete and challenging project: developing a fully automated pipeline to remove objects from a video. This topic was not only a perfect match for my interests but also aligned well with the lab's expertise.

With a clear objective established, I began an independent research phase to investigate the state-of-the-art methods and foundational concepts required for such a task. Through literature review and periodic discussions with my mentor, I determined that a modular pipeline approach would be the most effective strategy. As shown in Fig.2.1, the design is based on three distinct modules: an object detection module, a segmentation module, and finally, an inpainting module.

It is crucial to understand that this initial design served as a starting point. The pipeline evolved significantly as the project progressed and I gained a deeper understanding of the practical challenges. For example, the necessity of a robust preprocessing module to standardize video inputs and the integration of a tracking algorithm to handle object occlusions became evident during development. Each module presented its own set of difficulties, which required iterative problem-solving.

The following sections will first present the theoretical background necessary to grasp these challenges, with a particular emphasis on object detection, where I dedicated the majority of my efforts. Subsequently, I will detail the final architecture of the pipeline and the implementation of each of its components.

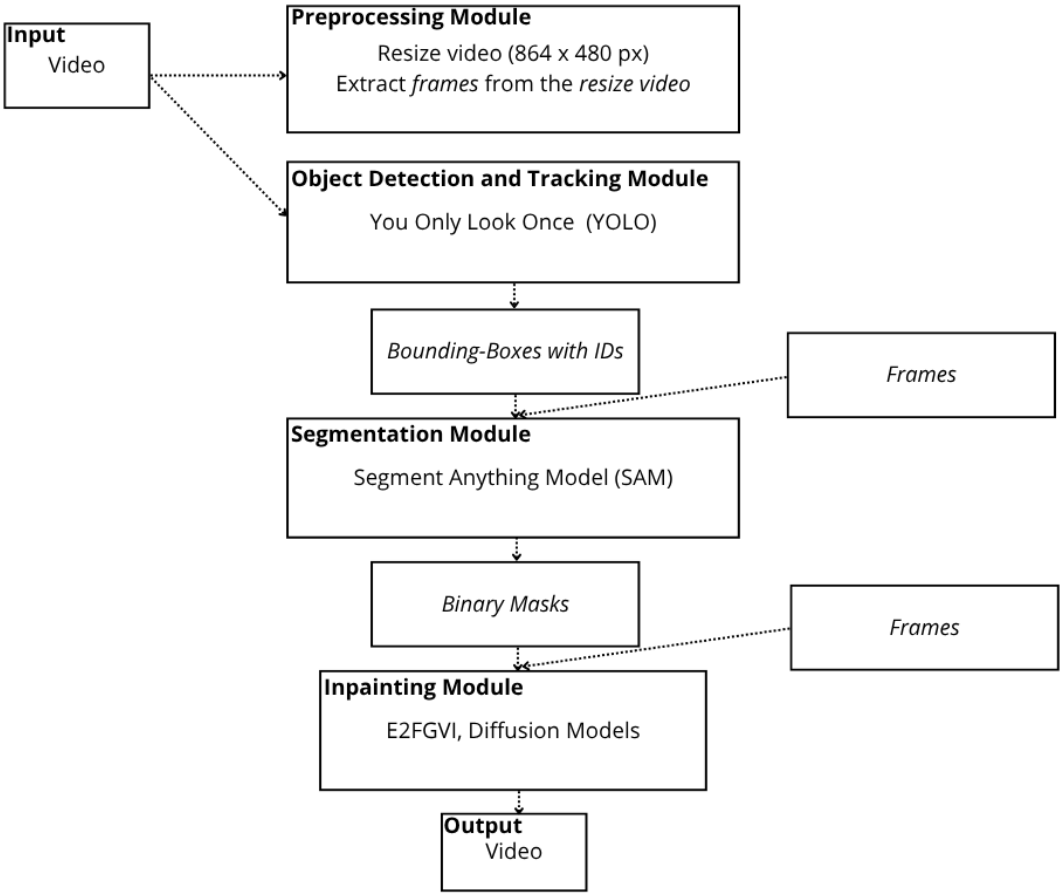


Figure 2.1: Presentation of the implemented video inpainting pipeline

2.1 Theoretical Background and State of the Art

2.1.1 Object Detection

I will develop the section on object detection more thoroughly than those on image segmentation and video inpainting, as it is the part where I spent the majority of my time and consequently faced the most challenges. Therefore, I will provide a detailed explanation of bounding boxes and their coordinate systems, as these were among the key difficulties I had to resolve.

General Principles of Object Detection

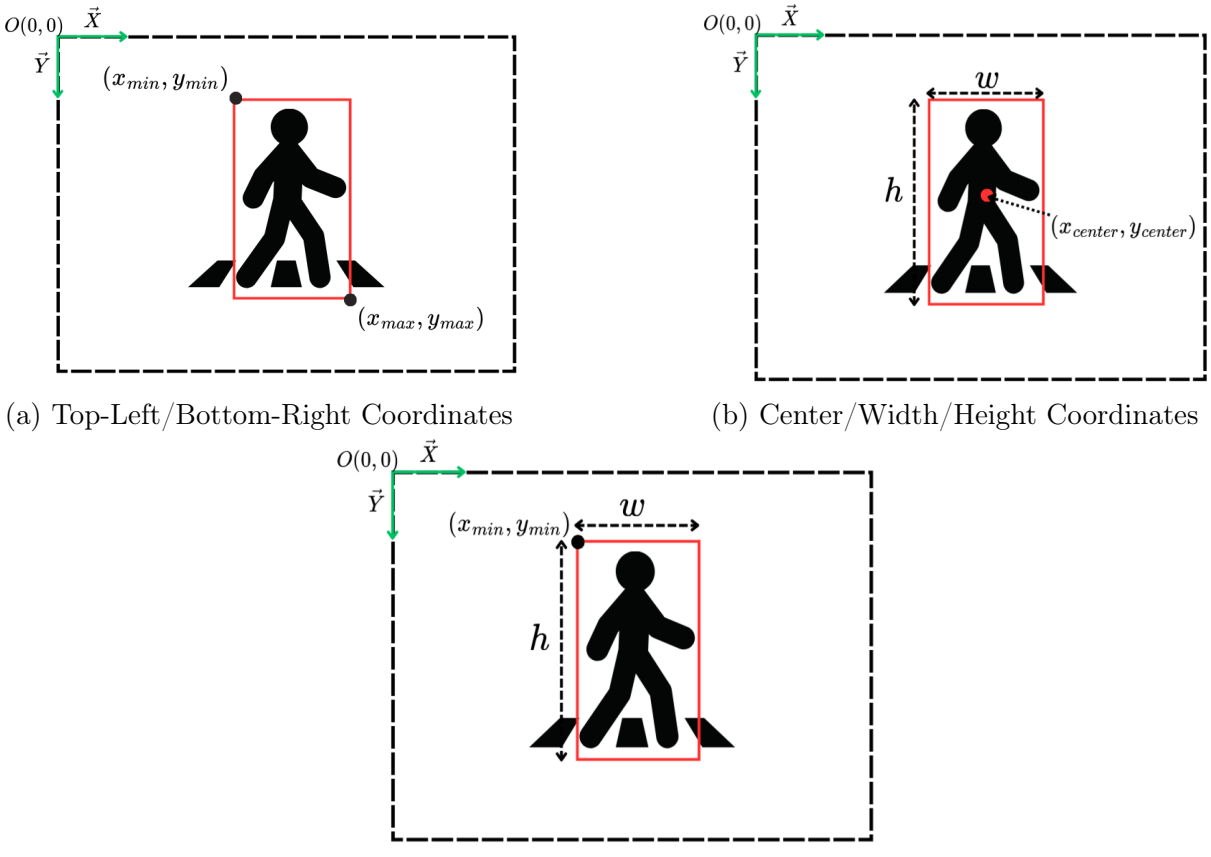
Object detection is a more advanced task than conventional image classification. While image classification aims to assign a single class label to an entire input image (determining if an image is a cat or a dog), object detection is a more advanced computer vision problem. It involves both identifying the presence of objects within an image and precisely localizing them, typically by drawing tight rectangular **bounding boxes** around each detected instance.

The typical formulation of an object detection problem is as follows:

- **Input:** An image or a sequence of video frames.
- **Classes:** A predefined set of N target classes, denoted as $C = \{C_1, C_2, \ldots, C_N\}$.
- **Output:** For each detected object belonging to one of the specified classes, the system provides:
  - Its predicted **class label** (C_i).
  - Its precise location within the image, defined by **the coordinates of a bounding box**.
  - A **confidence score**, indicating the model's certainty about the presence and class of the detected object within that bounding box.

Understanding Bounding Boxes

Bounding boxes are a fundamental component of object detection. As I previously mentioned, they are represented by coordinates.



(a) Top-Left/Bottom-Right Coordinates

(b) Center/Width/Height Coordinates

(c) Top-Left/Width/Height Coordinates

Figure 2.2: Different common formats for defining bounding box coordinates, illustrating how objects are localized within an image.

There are three primary types of coordinate formats used for bounding boxes:

- $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$: This format defines the bounding box by specifying the coordinates of its top-left corner $(x_{\min}, y_{\min})$ and its bottom-right corner $(x_{\max}, y_{\max})$. It clearly delineates the rectangular area occupied by the object (Fig 2.2a).
- $(x_{\text{center}}, y_{\text{center}}, w, h)$: In this format, the bounding box is defined by the coordinates of its center $(x_{\text{center}}, y_{\text{center}})$, along with its width (w) and height (h) (Fig 2.2b).
- $(x_{\min}, y_{\min}, w, h)$: This format combines the coordinates of the top-left corner $(x_{\min}, y_{\min})$ with the width (w) and height (h) of the box (Fig 2.2c).

When defining bounding boxes for object localization, the coordinate values can be expressed using either **absolute coordinates** or **normalized coordinates**. The choice between these two representations is crucial as it impacts how you handle images of different resolutions and how well your computer vision models generalize.

Absolute coordinates

- **Definition:** directly reference pixel locations within an image. For an image with a specific width and height in pixels, these coordinates correspond to the exact column and row numbers. If an image has dimensions W (width) and H (height) in pixels, an absolute x-coordinate will range from 0 to $W - 1$, and an absolute y-coordinate from 0 to $H - 1$.
- **Pros:**
  - **Intuitive for Display:** They are straightforward to use when drawing bounding boxes directly onto an image, as they map directly to pixel positions.
  - **Precision:** Provide exact pixel-level localization.
- **Cons:**
  - **Resolution Dependence:** This is their most significant limitation. If the image resolution changes (e.g., you resize a 640×480 image to 320×240), the absolute coordinates become incorrect. You must proportionally scale these coordinates to match the new image dimensions, which adds complexity.
  - **Not Robust for Models:** Training computer vision models, especially deep learning models, on datasets with images of varying resolutions becomes challenging with absolute coordinates, as the input scale for the bounding boxes changes with each image.

Normalized coordinates

- **Definition:** Each coordinate (or dimension like width/height) is divided by the corresponding image dimension. Normalized coordinates express bounding box parameters as ratios or fractions relative to the image's overall width and height. These values typically fall within the range of 0 to 1.
  - $x_{\text{norm}} = x_{\text{abs}}/\text{Image Width}$
  - $y_{\text{norm}} = y_{\text{abs}}/\text{Image Height}$

  - $w_{\text{norm}} = w_{\text{abs}}/\text{Image Width}$
  - $h_{\text{norm}} = h_{\text{abs}}/\text{Image Height}$
- **Pros:**
  - **Resolution Independence (Scalability):** This is their primary advantage. If an image is resized, the normalized coordinates of the bounding box remain valid without any modification because they represent a proportion of the image
  - **Model-Friendly:** Most modern deep learning object detection frameworks and libraries prefer or even require bounding box coordinates to be normalized.
- **Cons:**
  - **Indirect Interpretation:** To visualize the bounding box or to understand its exact pixel dimensions, you need to know the actual width and height of the image to convert the normalized coordinates back to absolute pixel values.

Importance in Computer Vision

Object detection has many direct real-world applications. For instance:

- **Autonomous Vehicles and Advanced Driver-Assistance Systems (ADAS)**: object detection is crucial for self-driving cars to perceive their environment. It allows them to identify and track other vehicles, pedestrians, cyclists, traffic signs, traffic lights, and road obstacles in real-time. This enables features like collision avoidance, adaptive cruise control, and automatic emergency braking.

- **Security and Surveillance**: In smart surveillance systems, object detection can automatically identify suspicious activities, such as detecting intruders in restricted areas, recognizing abandoned objects in public spaces (e.g., airports, train stations), or even identifying individuals of interest based on facial recognition or specific attributes. It significantly enhances the efficiency of monitoring and threat detection.

- **Specific case**: During my search for a work-study contract, I had several job interviews with the company ONEEX, which specializes in detecting counterfeit paper. I solved a serious game where one of my tasks involved using object detection to extract specific information from ID papers, such as the **Optical Character Recognition (OCR)** band, face, and flags. This was a direct and practical application of what I am currently learning and doing during my internship.

Another important insight is **the necessity of object detection as a prerequisite for more advanced computer vision concepts**. This was typically the case with this computer vision pipeline. I needed a high-performance object detection model to ensure proper inputs for the segmention module.

Evolution of object detection Models

Initially, object detection models were based on feature-based approaches that relied on handcrafted features. However, since approximately 2010, traditional methods gradually shifted towards deep learning approaches. Presently, object detection models can be broadly categorized into two main types: Two-stage Detectors and Single-stage Detectors.

Two-stage Detectors

Two stages detectors were the first models developped using Deep learning. The main idea behind this kind of models is to perform image classification models on different location in this image. The two main stages in such models are :

- **Region proposal** : The aim of this stage is to find where to perform the classification model on the image. To sum up, the first way to do this was to use a sliding window while more advanced models **Region-based Convolutional Neural Network (R-CNN)** [1] use region proposal to generate potential bounding boxes in an image and then run a classifier on these proposed boxes.
- **Classification and Bounding Box Refinement**: Once the potential bounding boxes are proposed, the second stage takes over. For each proposed region, features are extracted from the feature map. These features are then fed into a classification head to determine the class of the object within the box (e.g., person, car, or background). Simultaneously, a regression head refines the coordinates of the bounding box to more accurately localize the object. This two-step process allows for high accuracy in object detection.

Single-stage Detectors

Single-stage detectors are a class of object detection architectures that perform object localization and classification in a single forward pass of the network. Unlike two-stage detectors (e.g., Faster R-CNN), which first generate region proposals and then classify them, single-stage models directly predict bounding boxes and class probabilities over a dense sampling of possible locations. Key Characteristics of Single-Stage Detectors:

- **Speed:** Extremely fast inference thanks to the unified architecture. Ideal for real-time video or embedded systems.
- **Simplicity:** End-to-end training and fewer components compared to two-stage detectors.

Initial Challenges and Solutions

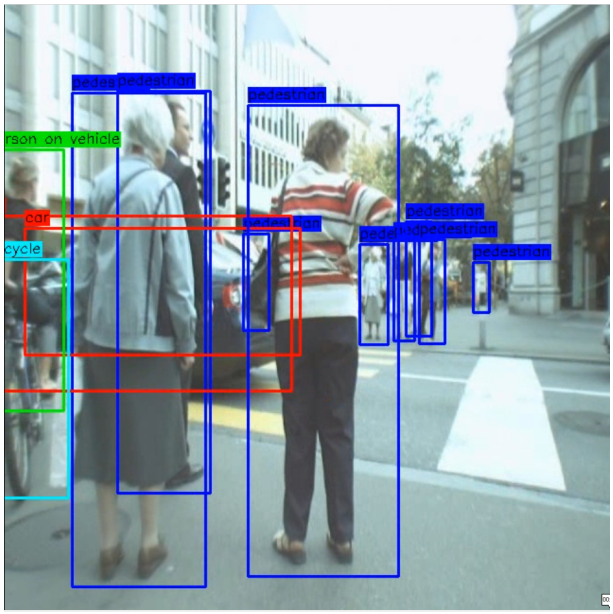


Figure 2.3: Visualizing object detection output on a street scene.

As shown in Fig. 2.3, object detection in real-world scenarios presents several inherent difficulties. The two most prominent challenges I identified were:

- **Scale Variation**: Objects can appear at vastly different scales, from small, distant figures to large, close-up individuals. As seen in Fig. 2.3, some pedestrians are far from the camera and difficult to detect, while others are much closer.
- **Occlusion**: Objects are frequently hidden, either partially or fully, by other objects, which complicates their detection. The car partially obscuring pedestrians in Fig. 2.3 is a clear example of this issue.

To address the occlusion problem, I decided to enhance the object detection process by combining it with **object tracking**. Tackling the scale issue proved more complex. The most effective strategy was to train a detection model on our own challenging dataset featuring significant scale variations. This approach is detailed further in the section 2.2, *Person Detection*, as developing a model robust to scale was a crucial requirement for the project's success.

I also encountered significant difficulties with the various **bounding box coordinate systems**. These challenges are detailed in subsubsection 2.2.3, *Difficulties encountered to create a dataset.*

Object Tracking

Object tracking is a fundamental task in computer vision that aims to **maintain consistent identities of detected objects across consecutive video frames**. In the context of this project, tracking was essential to associate and group all bounding boxes corresponding to the same individual over time, thereby enabling coherent downstream tasks such as segmentation and inpainting.

It also helps to address the occlusion problem. Indeed, if an individual is detected and then becomes partially hidden, object tracking algorithms can predict their continued presence and location based on their prior movement, ensuring they are still tracked despite the temporary obstruction.

To perform this task, I used the **BoT-SORT** [2] tracker. It is an enhanced multi-object tracking algorithm designed to improve robustness in challenging scenarios such as occlusions, overlapping detections, and identity switches. **BoT-SORT** builds upon the high-performance **ByteTrack** [3] framework and incorporates a re-identification module inspired by **DeepSORT** [4], offering a solid compromise between tracking accuracy and computational efficiency.

BoT-SORT works by combining:

- **Kalman filter:** Used to predict the future position of each object, assuming a constant motion model.
- **Appearance model (Re-ID):** A deep neural network extracts appearance features from object crops, allowing to match detections based not only on position but also on visual similarity.
- **Matching Strategy:** BoT-SORT employs a two-stage association process: it first matches tracks and detections using both spatial overlap (IoU) and appearance similarity, then refines associations to resolve ambiguities and handle low-confidence detections.

2.1.2 Image Segmentation

General Principles of Image Segmentation

Image segmentation is a computer vision task that involves dividing an image into meaningful parts to simplify its analysis. Unlike object detection, which provides bounding boxes around objects, segmentation assigns a label to each individual pixel, enabling a more precise understanding of object shapes and boundaries.

There are two main types of segmentation:

- **Semantic Segmentation:** Assigns a class label (e.g., "person", "car", "tree") to each pixel, without distinguishing between different instances of the same class.
- **Instance Segmentation:** Combines object detection and semantic segmentation by identifying and segmenting each individual object instance in the image.

To build the pipeline, I used **instance segmentation** as I wanted to obtain masks that could be used as input to an inpainting model. The goal was to identify and isolate individual objects (in this case, people) in order to remove or modify them in the scene via image inpainting.

Concerning the masks, we can distinguish two main types of masks:

- **Binary masks:** These are two-dimensional arrays of the same size as the original image, where each pixel is either 0 or 1. A value of 1 indicates that the pixel belongs to the object of interest, and 0 otherwise. These masks are crucial for tasks like inpainting, where we want to clearly define the region to be removed or altered.
- **Categorical masks:** Instead of representing only one object, these masks encode the object category for each pixel (e.g., 1 = person, 2 = car, etc.). In semantic segmentation, a single mask covers all pixels in the image, with each pixel labeled according to the class it belongs to. However, this format does not separate different instances of the same class, making it less suitable for tasks like selective inpainting.

In my case, **binary masks** are the most appropriate choice, as I needed to remove specific individuals from the image. By using instance-level binary masks, the pipeline is able to accurately isolate each person and pass their corresponding mask to the inpainting model, allowing for controlled and localized modifications in the image.

Importance and Applications

- **Medical Imaging:** Segmenting individual cells, tumors, or organs.
- **Autonomous Driving:** Differentiating pedestrians, vehicles, or traffic signs for safe navigation.
- **Agricultural Monitoring:** Counting and analyzing individual plants, fruits, or pests.
- **Augmented Reality:** Interacting with segmented real-world objects for overlay precision.

2.1.3 Video Inpainting

Concept of Video Inpainting

Video inpainting consists of restoring or removing objects in a video sequence by filling missing regions in a way that is **spatio-temporally consistent**. This implies preserving both the visual coherence within individual frames (**spatial consistency**) and the continuity across consecutive frames (**temporal consistency**).

Video inpainting is a challenging problem:

- **Temporal consistency:** The inpainted content must be visually smooth across consecutive frames to avoid flickering artifacts.
- **Complex motion:** Moving objects and backgrounds require advanced techniques to propagate content accurately over time.
- **Multiple occlusions:** Masked regions may cover dynamically changing objects, complicating restoration.

Most of inpainting models are basec on the following methods:

1. **Flow-guided inpainting:** Optical flow estimation methods are used to track motion between frames, allowing the propagation of known pixel content into masked regions temporally.
2. **Deep generative models:** Approaches such as spatio-temporal **Generative Adversarial Networks (GANs)** and **Transformers** are used to generate missing video content by learning spatial patterns and temporal dependencies from large-scale video datasets. These models are capable of synthesizing realistic frames that are consistent both visually and temporally
3. **Patch-based and non-local methods:** These algorithms copy patches from known regions either within the same frame or across frames, exploiting self-similarities in space and time.

Applications and Challenges

- **Video editing and post-production:** Removing unwanted objects, logos, or watermarks from videos while maintaining seamless visual continuity.
- **Visual effects (VFX):** Generating realistic backgrounds or filling missing frames in special effects pipelines.
- **Augmented reality (AR) and virtual reality (VR):** Enhancing immersive experiences by removing occlusions or reconstructing scenes in real-time.
- **Surveillance and security:** Recovering occluded or corrupted video data for analysis.
- **Restoration of old or damaged footage:** Filling in missing or corrupted frames to improve video quality.

With the theoretical framework now established, the following sections will present the model selection process and discuss the challenges encountered at each development stage.

2.2 Person Detection

2.2.1 You Only Look Once (YOLO)

YOLO [5] is a single-stage object detector initially developed by Joseph Redmon other researchers in 2016. Unlike traditional two-stage detectors such as R-CNN, which first generate region proposals and then classify them, YOLO directly predicts bounding boxes and class probabilities from the full image in a single forward pass of a convolutional neural network. This unified architecture allows for real-time object detection with high speed and competitive accuracy.

YOLO: Key Features and Limitations

This section is based on the first paper published on YOLO in 2016. Since, a lot of improvement has been done. However, the main breaktrough concept which is to adress the object detection problem as a regression is still the same.

- **Single, End-to-End Network:** YOLO uses a single convolutional neural network that directly predicts multiple bounding boxes and their associated class probabilities from the full image in one evaluation. This means the entire detection pipeline is a single network that can be optimized end-to-end, leading to a more efficient and effective process.
- **Global Reasoning:** Unlike prior methods, YOLO sees the entire image during both training and testing. This allows it to implicitly encode contextual information about objects and their surroundings. As a result, YOLO is significantly less likely to predict false positives on background compared to methods like Fast R-CNN.
- **Exceptional Speed:** Because it is a single, unified network, YOLO is extremely fast. The base YOLO model can process images in real-time at 45 frames per second. This enables real-time processing of streaming video with very low latency.
- **Generalizable Representations:** YOLO learns very general representations of objects. When trained on natural images and tested on new domains like artwork, YOLO outperforms other top detection methods by a wide margin. This means it is less likely to fail when encountering new or unexpected inputs.

- **Simplified Design:**

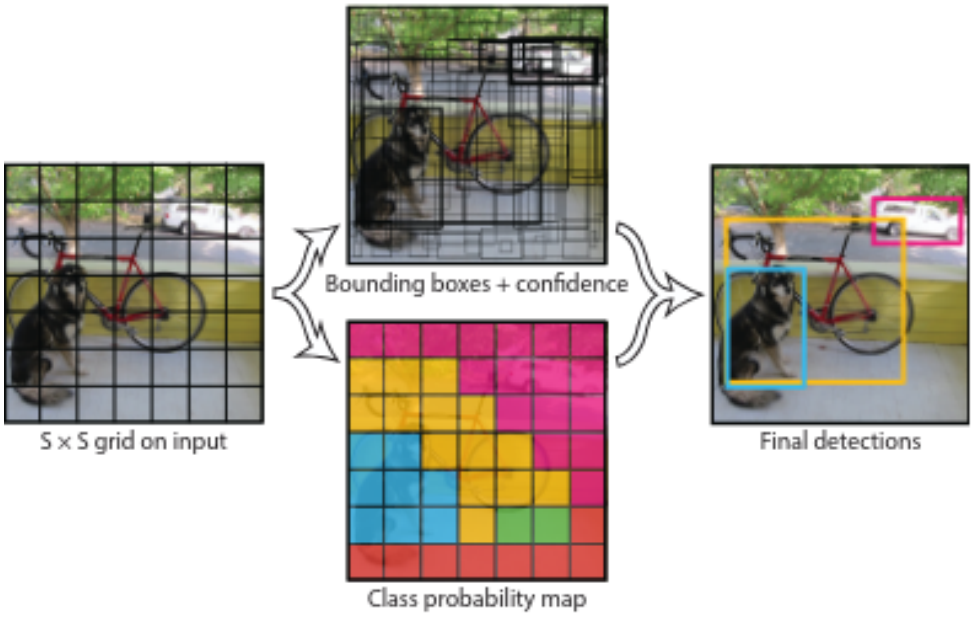


Figure 2.4: Conceptual model of YOLO

The system works by dividing the input image into a grid (e.g., 7×7) (Fig. 2.4). Each grid cell is responsible for detecting objects whose center falls within it. Each cell predicts a set number of bounding boxes, confidence scores (how likely a box contains an object and how accurate it is), and conditional class probabilities (what object class it is, if an object is present). All these predictions are made simultaneously by the single network.

- **Smart Loss Function:** YOLO optimizes a multi-part loss function that directly corresponds to detection performance. It is designed to give more weight to errors in bounding box coordinates and less to confidence predictions for cells that do not contain objects, addressing stability issues seen in simpler sum-squared error functions.

While YOLO brought significant advancements in speed and generalization, it did have some limitations: it struggled to precisely localize some objects, especially small objects that appear in groups (like a flock of birds), and might not generalize well to objects with very unusual aspect ratios or configurations.

Why Choosing YOLO?

- **Faster than Two-Stage Detectors:** YOLO is ideal for real-time applications. Even though I did not implement real-time data acquisition, I considered this choice relevant for potential future improvements in that direction. While single-stage detectors are traditionally seen as slightly less accurate than two-stage ones, recent YOLO versions have significantly narrowed that gap to the point where the difference is barely noticeable.

- **Recommended by Experts:** During discussions on object detection with PhD students from the CVL, YOLO was frequently recommended due to its balance of performance and simplicity.

- **Ease of Deployment:** A noteworthy point is the fragmented development of YOLO since version 3. The original author discontinued his work, and several independent teams have since continued development, resulting in multiple "lineages" of YOLO models.

  One of the most prominent branches is maintained by the company **Ultralytics**, which specializes in deploying state-of-the-art computer vision models. Ultralytics continues its own YOLO development, known for being user-friendly, actively maintained, and based on the widely adopted PyTorch framework.

  For this project, I exclusively used models from the Ultralytics lineage. At the time of writing, their most recent major version is YOLOv12. However, the pipeline was developed using YOLOv11.

2.2.2 Train YOLO with a custom dataset

YOLO models deployed by Ultralytics are pre-trained on the **Microsoft Common Objects in Context (COCO)** [6] dataset, a widely adopted benchmark in computer vision. COCO is a large-scale dataset designed for object detection, instance segmentation, and image captioning. It contains over 330,000 images, of which more than 200,000 are labeled, encompassing more than 80 object categories.

Pre-trained YOLO models based on COCO exhibit strong generalization capabilities across a variety of natural scenes. However, these models are optimized for the general domain of COCO, where objects typically occupy a significant portion of the image and are well-distributed across categories.

In this project, we are dealing with highly specific image characteristics: individuals can appear small, partially occluded, or located in dense clusters. This domain shift leads to a performance degradation when using off-the-shelf pre-trained models. To bridge this gap, it becomes crucial to fine-tune the YOLO model on a custom dataset that mirrors our target distribution. Starting from a YOLO pre-trained model accelerates convergence and helps preserve robust low-level features (e.g., edge, color, texture filters), while enabling the network to specialize on our domain-specific patterns.

This is why I decided to fine-tune the YOLO model on a custom dataset tailored to the specific characteristics of the task. The following sections describe in detail the dataset creation process, the training pipeline, and an in-depth analysis of the results obtained after fine-tuning.

2.2.3 Creating the dataset

The Multiple Object Tracking Challenge

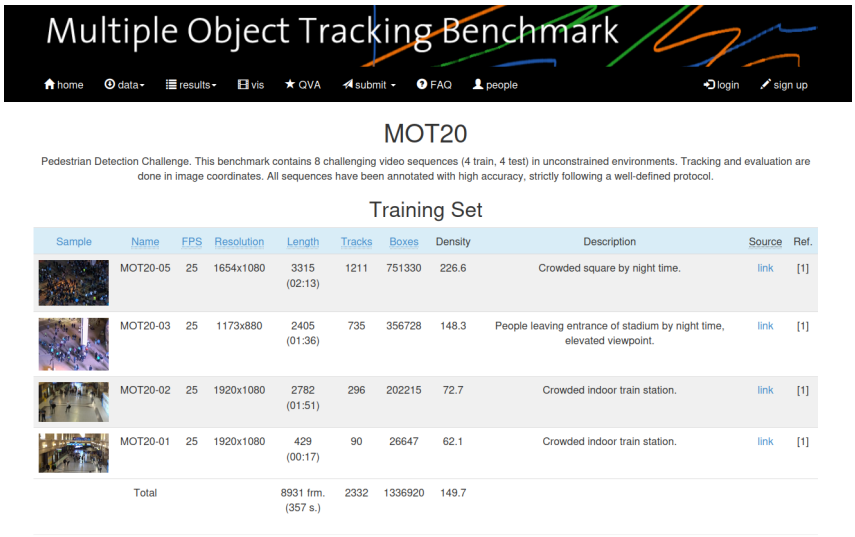


Figure 2.5: MOT20 training dataset on the MOTchallenge website

The dataset used in this project was primarily built from the video sequences provided by the **Multiple Object Tracking Challenge (MOTChallenge)** (see Fig.2.5). Launched in 2015 by leading figures in the Computer Vision community, the MOTChallenge was created to standardize the evaluation of multi-object tracking algorithms and foster reproducible research in the field.

The website hosts various challenges in object detection and tracking. Each challenge provides standardized datasets composed of annotated videos captured in urban environments. These annotations include pedestrians, vehicles, and other road users, each labeled with bounding boxes and unique IDs.

These datasets were particularly well-suited to my project as they offer real-world street camera imagery combined with precise human annotations which is essential for training a robust person detection model. This was especially valuable given the lack of purpose-built datasets for ethically sensitive applications.

To construct my dataset, I selected sequences from both the **MOT17** (an improved version of **MOT16** [7]) and the **MOT20** datasets. Both focus specifically on pedestrian detection and tracking. Each video sequence is provided with:

- A folder containing **all video frames** (as individual images),
- A corresponding **annotation file** detailing object IDs, bounding boxes, and frame-level metadata.

Difficulties encountered to create a dataset

The creation of the dataset proved to be a challenging and time-consuming task, occupying a significant portion of my efforts. The main difficulty stemmed from the structure of the provided annotation file. Figure 2.6 illustrates the first few lines of the ground truth file associated with a video sequence from the MOT20 dataset.

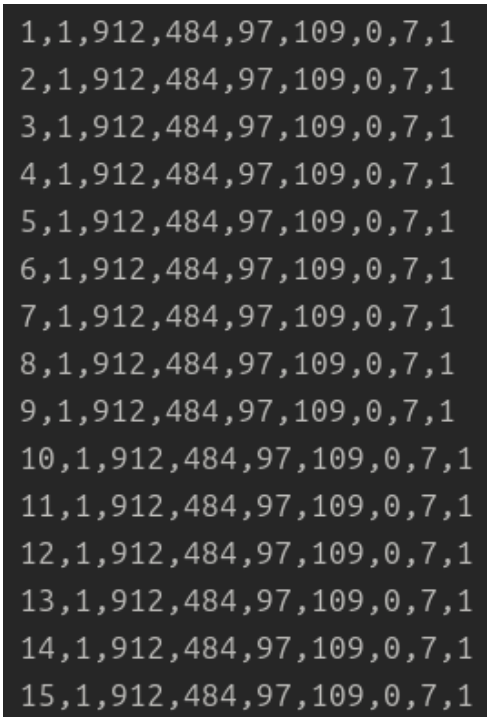


Figure 2.6: Excerpt from a ground truth file of a video sequence in the MOT20 dataset

As explained in the MOT16 paper, each column of the annotation file has a precise and standardized meaning. Table 2.1 presents the full column definitions, slightly adapted to fit our specific case and notation.

Pos.	**Name**	**Description**
1	Frame	Frame index where the object appears.
2	ID	Unique identifier for each pedestrian trajectory.
3	x_{min}	X-coordinate of the top-left corner of the bounding box.
4	y_{min}	Y-coordinate of the top-left corner of the bounding box.
5	w	Bounding box width (in pixels).
6	h	Bounding box height (in pixels).
7	Flag	Indicates if the entry is valid (1) or ignored (0).
8	Class	Object type (e.g., pedestrian, vehicle).
9	Visibility	Ratio between 0 and 1, quantifying the visible portion of the object.

Table 2.1: Description of ground truth file columns (MOT format).

The main difficulties I encountered were primarily due to the annotation file format, but not exclusively:

- **Single annotation file instead of multiple:** YOLO expects a separate annotation file for each frame. The MOT format, however, provides a single '.txt' file per sequence, containing the annotations for all frames. This required a complete restructuring of the annotation logic.

- **Bounding box coordinates format:** The original annotations use the Top-Left / Width / Height format (see Fig. 2.2c), whereas YOLO requires normalized Center X / Center Y / Width / Height. The conversion added complexity, especially in a context where image resizing was also applied.

- **Image size and resolution:** The raw video frames were mostly in Full HD (1920×1080), which was computationally expensive for downstream tasks such as image segmentation. While YOLO can internally resize images during training, I chose to pre-process and downscale the images to 864×480. This made coordinate transformation trickier, as I had to factor in scaling when computing the normalized bounding boxes.

- **Excessive number of frames per sequence:** Initially, I believed that a higher frame count would improve model training. However, I later realized that due to high video framerates, many consecutive frames were almost identical (e.g., frame 5 and frame 15). This redundancy offered little added value. More importantly, I discovered, somewhat late, that dataset diversity (e.g., in weather conditions, lighting, or camera angles) contributes far more to training robustness than sheer volume.

- **Class Mapping and Filtering:** In the MOT datasets, class indices start from 1 and go up to 12, whereas YOLO requires class indices to start from 0. Additionally, a second challenge was related to the relevance of the classes: many classes in the MOT datasets were either not useful or needed to be grouped. For instance, classes such as *Car* or *Static Occluder* had to be discarded, while others like *Pedestrian*, *Person on Vehicle*, and *Static Person* were merged into a single unified class representing all person-like instances.

To address these issues, I developed **a Python script designed to automate the conversion of MOT-formatted datasets into YOLO-compatible format**. The objective was to be able to launch the transformation process via a single command line instruction. However, due to the large number of parameters involved, modifying them directly in the command line would have been tedious and error-prone. I therefore opted for a configuration-based approach using a JSON file, as illustrated in Fig. 2.7.

```
{
    "PathToDataFolders": "/home/mapicasse/Documents/02_Academic/Internship/YOLO11_SAM_E2FGVI/dataset/raw/final_MOT_dataset/train",
    "Folders": {
        "MOT17-02": [1920, 1080],
        "MOT17-05": [640, 480],
        "MOT17-09": [1920, 1080],
        "MOT17-13": [1920, 1080],
        "MOT20-01": [1920, 1080],
        "MOT20-02": [1920, 1080],
        "MOT20-05": [1654, 1080]
    },
    "OutputDir": "/home/mapicasse/Documents/02_Academic/Internship/YOLO11_SAM_E2FGVI/training/train",
    "Classes": {
        "1": "person",
        "2": "person",
        "7": "person"
    },
    "ResizeImages": true,
    "TargetSize": [854,480],
    "SubsampleRate": 7
}
```


Figure 2.7: Example of a JSON configuration file for converting MOT datasets into YOLO training format.

Below is a description of the configuration parameters used:

- **PathToDataFolders**: Path to the main directory containing the subfolders for each video sequence.
- **Folders**: Dictionary where each key is the name of a subfolder (i.e., a video sequence), and each value is a list specifying the corresponding target image size `[width, height]`.
- **OutputDir**: Path to the output directory. For each video sequence, a subfolder is created containing two subdirectories: `img/` (resized frames) and `annotations/` (YOLO-formatted label files, one per frame).
- **Classes**: Dictionary mapping original MOT class IDs to the new class names used in YOLO. Unlisted classes are ignored.
- **ResizeImages**: Boolean flag indicating whether or not to resize images.
- **TargetSize**: Target dimensions for resizing images (format: `[width, height]`).
- **SubsampleRate**: Controls frame subsampling to reduce dataset redundancy. For example, a value of 7 means every 7th frame is selected (`frame 0, 7, 14, ...`), allowing for greater scene diversity and reduced training time.

The `Classes` turned out to be **a critical element** in the dataset conversion process. I realized after several days of training that I had made a major omission: the `Person on Vehicle` class (class index 2 in the MOT dataset) had not been included in the selected classes during annotation conversion. As a result, any instance of a person riding a bicycle or motorcycle detected by the model was wrongly considered a false positive, since this class had been excluded from the generated ground truth files. In reality, this class should have been grouped with `Pedestrian` and `Static Person` to form a unified "`person`" class. This issue highlights the importance of thoroughly reviewing and configuring class mappings when converting datasets, especially when moving from a complex annotation format like MOT to a simpler one like YOLO.

2.2.4 Presentation of the final Dataset

Once python script was working, I was finally able to build the dataset. I tried to have a wide variety for the training and to evaluate the models on some challenging sequence. However, I had only 11 video sequences to build the dataset. It was not easy to make some choice not to baised the results.

Training set

Table 2.2: Training video sequences

Training video sequences						
Name	**Resolution**	**Frames**	**Camera**	**Viewpoint**	**Conditions**	**Scene**
MOT17-02	864×480	86	static	medium	cloudy	large square
MOT17-05	864×480	93	moving	medium	sunny	street scene
MOT17-09	864×480	75	static	low	indoor	Main aisle in a mall
MOT17-13	864×480	84	moving	high	sunny	busy intersection
MOT20-01	864×480	86	static	high	indoor	train station
MOT20-02	864×480	93	static	high	indoor	train station
MOT20-05	864×480	95	static	high	night	square
Total		**612**				

(a) MOT17-02

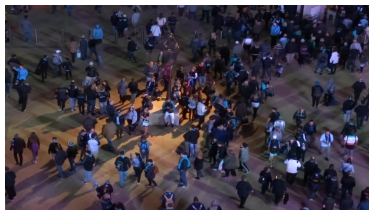
(b) MOT17-05

(c) MOT17-09

(d) MOT17-13

(e) MOT20-01

(f) MOT20-02



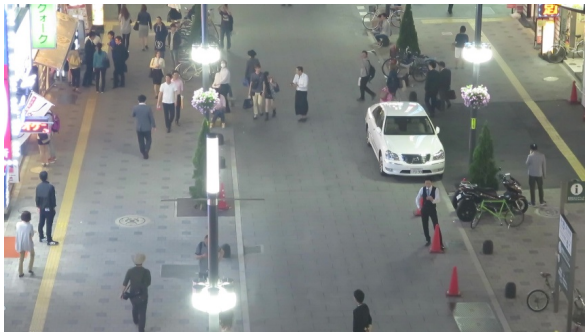
(g) MOT20-05

Figure 2.8: Example frames from each MOT sequence

Validation set

Table 2.3: Validation video sequences

Validation video sequences						
Name	**Resolution**	**Frames**	**Camera**	**Viewpoint**	**Conditions**	**Scene**
MOT17-04	864×480	88	static	high	night	Pedestrian street
MOT17-10	684×480	82	moving	medium	night	pedestrian street
Total		**170**				



(a) MOT17-04

(b) MOT17-10

Figure 2.9: Example frames from different MOT sequences

Test set

Table 2.4: Test video sequences

Validation video sequences						
Name	**Resolution**	**Frames**	**Camera**	**Viewpoint**	**Conditions**	**Scene**
MOT17-11	864×480	90	moving	medium	indoor	mall aisle
MOT20-03	684×480	93	static	high	night	stadium entrance
Total		**141**				

(a) MOT17-11

(b) MOT20-03

Figure 2.10: Example frames from different MOT sequences

Concerning Tables 2.2, 2.3, and 2.4, one important detail is the *Camera* column, which indicates whether the camera was *static* or *moving* during capture. At first glance, this distinction may seem irrelevant, since the model is trained on individual frames without exploiting temporal coherence. However, it implicitly introduces diversity in viewpoints and scene dynamics: sequences with moving cameras provide a wider range of perspectives and parallax, whereas static cameras yield more consistent framing. Figures 2.8, 2.9, and 2.10 illustrate representative scenes from each dataset.

2.2.5 Training parameters

To train the model I mainly used these parameters :

- **Model:** *YOLOv11n.pt*, a lightweight pretrained YOLOv11 model.
- **Epochs:** 20 training epochs were conducted to ensure sufficient convergence while preventing overfitting.
- **Batch size:** 8, which provides a good trade-off between computational efficiency and gradient stability.
- **Device:** `cpu`. Training was performed on a CPU, which significantly increased training time compared to a GPU.
- **Mosaic:** Enabled. This technique, integrated into YOLO, combines four training images into a single one. This simulates diverse scene compositions and object interactions, proving highly effective for complex scene understanding. This parameter is particularly beneficial in our case due to the small dataset and the similarity of many images.

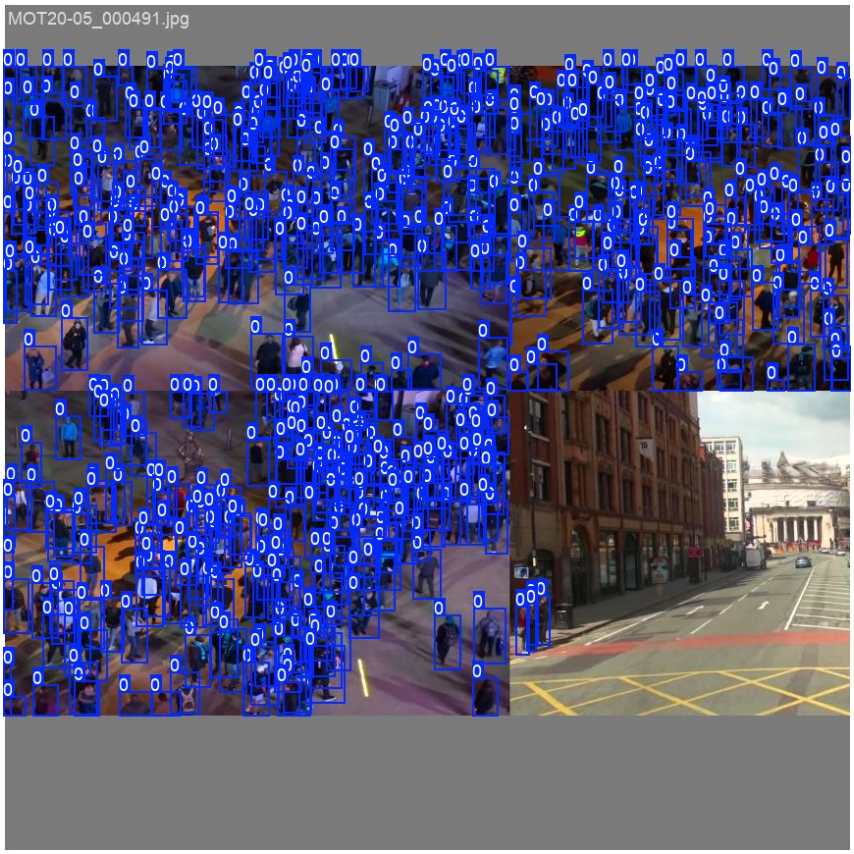


Figure 2.11: Example of mosaic augmentation during training.

Above, Figure 2.11 displays an example image from training batch 0, which is a direct result of the Mosaic augmentation.

2.2.6 Metrics

To evaluate the performance of the model, I mainly focused on the following metrics :

- **Precision:** The accuracy of the detected objects, indicating the proportion of correct positive predictions among all positive predictions. It is defined as:

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

  where TP is the number of true positives and FP is the number of false positives.

- **Recall:** Recall measures the proportion of actual objects that were correctly detected. It reflects the model's ability to find all relevant instances.

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

  where FN is the number of false negatives (i.e., missed detections).

- **F1-score:** The F1-score is the harmonic mean of precision and recall. It provides a single metric that balances both aspects, especially useful when there is class imbalance.

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

- **Intersection over Union (IoU):** It's a fundamental metric used in object detection to evaluate the accuracy of predicted bounding boxes. It quantifies the degree of overlap between the predicted bounding box and the ground truth bounding box.

  Formally, the IoU is defined as the ratio between the area of overlap and the area of union of the two bounding boxes:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} \tag{2.1}$$

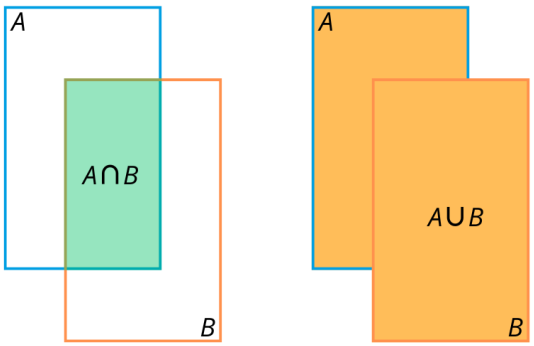


Figure 2.12: Illustration of the Intersection over Union (IoU) between two bounding boxes.

Figure 2.12 provides a clear illustration of this ratio. Bounding box A represents the ground truth, while bounding box B corresponds to the predicted bounding box. The green colored region represents the area of intersection between the two boxes, while the orange one represents the area of union including the total area covered by both.

This metric yields a value between 0 and 1:

- IoU = 0 indicates no overlap,
- IoU = 1 indicates a perfect match between prediction and ground truth.

A commonly used threshold to consider a detection as correct (true positive) is an **IoU greater than or equal to 0.5**, although stricter thresholds (e.g., 0.75 or 0.9) may be used in high-precision contexts.

- **Average Precision (AP):** The Average Precision is defined as the area under the Precision–Recall (PR) curve. To build the PR curve, we use the following method:

  1. an IoU_t threshold is first fixed
  2. For this IoU_t, the confidence score threshold s_i is varied over its range (typically from 0 to 1)
  3. For each s_i, the corresponding $(\text{Precision}_i, \text{Recall}_i)$ pair is computed, providing one point of the curve

  The sequence of these pairs forms the PR curve, from which the AP is obtained by integrating the precision as a function of recall

- **mean Average Precision (mAP):** The mean Average Precision is computed as the mean of the AP values over all target classes:

$$\text{mAP} = \frac{1}{C} \sum_{c=1}^{C} \text{AP}_c$$

  where C is the total number of classes and AP_c is the Average Precision for class c.

  Two main variants are commonly used:

  - **mAP50**: computed with a fixed IoU threshold of 0.5.
  - **mAP50-95**: computed as the average of mAP values over IoU thresholds ranging from 0.5 to 0.95 with a step of 0.05.

  These two variants are the ones used to evaluate the model's performance in Ultralytics. The **mAP50-95** metric is specifically used as the primary criterion to select the best-performing model at the end of training.

2.2.7 Results

Training and Validation Set Results

The training and validation curves from the YOLOv11 model, trained on a custom dataset, reveal important insights into its learning process. Figure 2.13 displays these curves, highlighting both expected and unusual behaviors, particularly within the validation set metrics.

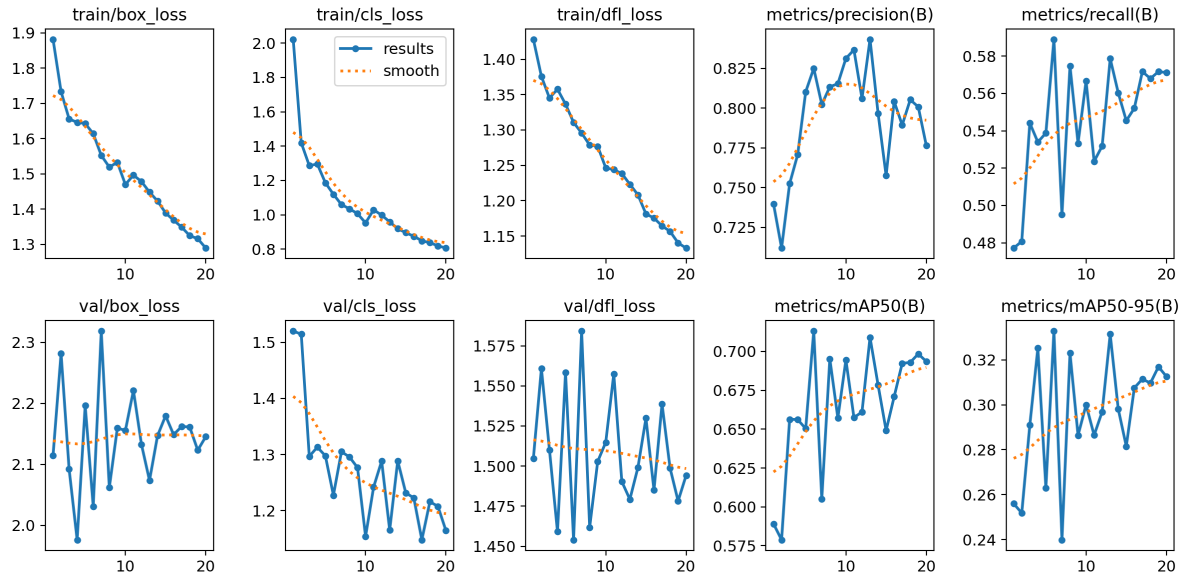


Figure 2.13: Training and Validation Curves for YOLOv11 Model. The top row shows training loss curves (box, classification, DFL), and validation metrics (precision, recall). The bottom row displays validation loss curves (box, classification, DFL) and validation metrics (mAP50, mAP50-95).

Curves Analysis:

- **Training Loss Curves:** The training losses (`train/box_loss`, `train/cls_loss`, `train/dfl_loss`) show a consistent downward trend, indicating effective learning and error minimization throughout the training process. This is characteristic of a well-converging model.
- **Validation Box Loss (`val/box_loss`):** The curve remains relatively flat, showing limited improvement.
- **Validation DFL Loss (`val/dfl_loss`):** A slight decrease is observed, but the trend remains modest. This suggests minimal refinement in the predicted bounding box distributions.
- **Validation Classification Loss (`val/cls_loss`):** This curve exhibits a clear downward trend, indicating the model is progressively improving its classification performance on the validation set.
- **Oscillatory Behavior:** All validation loss curves display noticeable peaks and drops. These fluctuations are atypical in stable training and may point to validation instability, possibly due to sampling variability or data imbalance.

- **Validation metrics:** All validation metrics (`metrics/precision(B)`, `metrics/recall(B)`, `metrics/mAP50(B)`, `metrics/mAP50-95(B)`) generally show an increasing trend, which is positive, indicating improved performance on the validation set over time. However, similar to the validation loss curves, these metrics also exhibit pronounced fluctuations (peaks and drops). While the overall trend is upward, the instability implies that the model's performance on the validation set is not consistently improving in a smooth manner

The unusual behavior of the validation curves and metrics clearly indicated that further investigation was necessary. What particularly triggered this analysis was the test set performance, where the custom model significantly outperformed the base model.

Quantitative Results on the Validation Set

After training, the best model was selected based on the `mAP50-95` on the validation set. I compared its performance with the base YOLOv11n model. The results are presented in Table 2.5.

Table 2.5: Metrics on the validation set: best model vs. base model

Results on Validation Set					
Model	**Precision**	**Recall**	**F1-score**	**mAP50**	**mAP50-95**
Custom Model (Best)	0.820	0.589	0.685	0.713	0.333
YOLOv11n (Base)	0.789	0.575	0.661	0.694	0.400

Although the custom model performs slightly better on most metrics, the base model shows a higher `mAP50-95`, raising initial concerns about the robustness of the custom model. However, a deeper inspection revealed that the validation set was composed of highly challenging sequences (e.g., low light, occlusions, and dense scenes), which made it harder for both models to perform well and likely penalized the custom model's bounding box regression metrics.

Table 2.6: Metrics on the validation set: best model vs. base model

Results on Validation Set						
Model	Name	**Precision**	**Recall**	**F1-score**	**mAP50**	**mAP50-95**
Custom Model (Best)	MOT17-04	0.844	0.652	0.735	0.769	0.364
YOLOv11n (Base)	MOT17-04	0.829	0.593	0.691	0.726	0.457
Custom Model (Best)	**MOT17-10**	**0.751**	**0.652**	**0,698**	**0.568**	**0.265**
YOLOv11n (Base)	**MOT17-10**	**0.750**	**0.552**	**0,635**	**0.655**	**0.312**

I consider the MOT17-10 sequence to be particularly challenging. The camera is moving, which introduces motion blur. The video is shot at night, and many individuals appear far from the camera, making detection and localization more difficult. The custom model likely detects more difficult instances such as small, distant, or partially occluded persons, which are harder to localize precisely. As a result, many of these detections fail to meet the IoU thresholds required by `mAP50` and `mAP50-95` (see Fig. 2.6). This explains the lower scores on these metrics, even though the overall detection coverage improves.

This explains why the validation results may underestimate the true capabilities of the custom model. When evaluated on a separate test set with more balanced conditions, the custom model significantly outperformed the base model across all metrics, validating its generalization power.

Quantitative Results on the Test Set

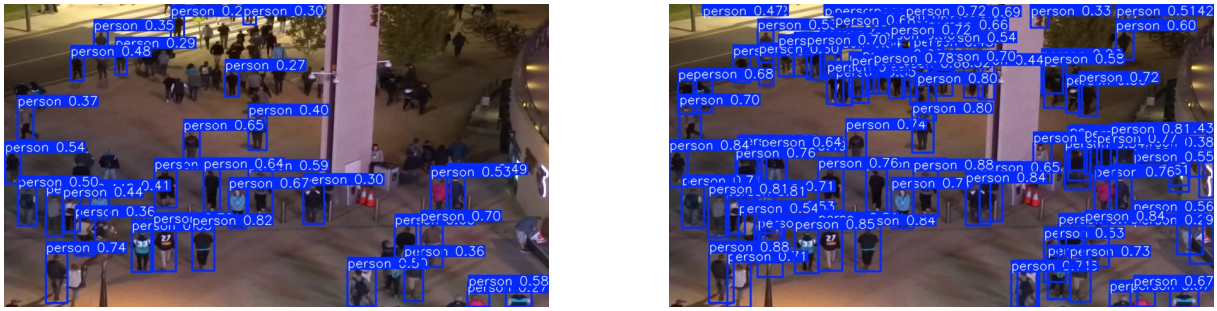
Table 2.7: Metrics on the test set: best model vs. base model

Results on Validation Set					
Model	**Precision**	**Recall**	**F1-score**	**mAP50**	**mAP50-95**
Custom Model (Best)	0.884	0.770	0.823	0.853	0.442
YOLOv11n (Base)	0.741	0.449	0.559	0.572	0.24

Table 2.8: Metrics on the test set: custom model vs. base model

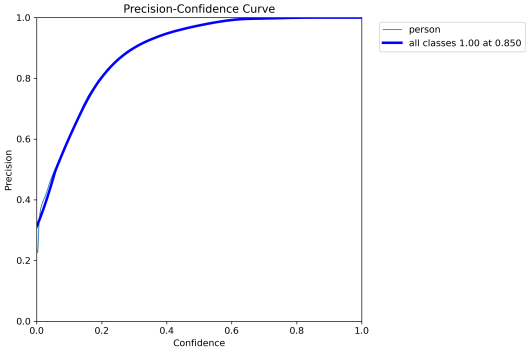
Results on Validation Set						
Model	Name	**Precision**	**Recall**	**F1-score**	**mAP50**	**mAP50-95**
Custom Model (Best)	MOT17-11	0.883	0.624	0.731	0.724	0.412
YOLOv11n (Base)	MOT17-11	0.914	0.631	0,746	0.703	0.496
Custom Model (Best)	MOT20-03	0.883	0.781	0,828	0.87	0.457
YOLOv11n (Base)	MOT20-03	0.699	0.498	0,581	0.6069	0.216

Tables 2.7 and 2.8 provide comprehensive insights into model performance across different scene configurations. Our custom model demonstrates strong adaptability, maintaining high precision and recall on both medium-view scenes (MOT17-11) and challenging crowded scenes (MOT20-03). In contrast, the YOLOv11n base model struggles significantly on MOT20-03, which features dense crowds and smaller object sizes, resulting in lower precision, recall, and mAP scores. This demonstrates that the custom model is capable of handling complex, crowded environments while maintaining strong performance on easier scenes.
Below, the Fig. 2.14 show direct predictions made betwwen the base model and the custom one. They are still some mistakes in the custom the performance has improved.

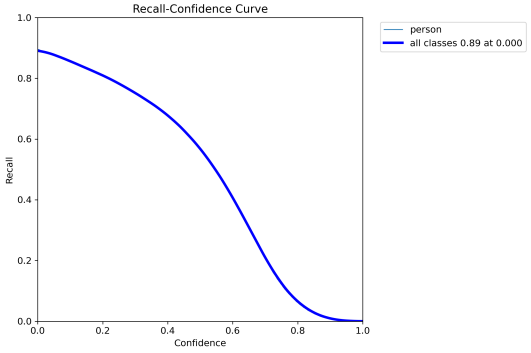


(a) Base model (b) Custom model

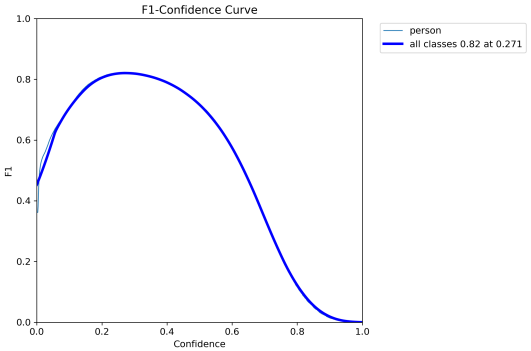
Figure 2.14: Comparison of the predictions made between the base and the custom model



(a) Precision vs Confidence



(b) Recall vs Confidence



(c) F1-score vs Confidence

Figure 2.15: Model evaluation curves as a function of confidence threshold.

Fig. 2.15 shows the evolution of precision, recall, and F1-score as functions of the confidence threshold. These curves are essential for selecting an appropriate value for the `conf` parameter during inference. According to Fig. 2.15c, a threshold of 0.271 yields the best trade-off between precision and recall, making 0.25 a reasonable default choice.

2.2.8 Implementation in the Pipeline

The integration of the detection module is based on the `DetectTrack` class, which implements the functionalities required to detect and track people throughout the video. The core idea is that `DetectTrack` combines both detection and tracking models, while most of the low-level operations are handled by Ultralytics. To perform detection and tracking, the pipeline relies on the `track` method of the `DetectTrack` class.

The attribute `_raw_detections` stores the tracking results frame by frame. To facilitate the subsequent masking step, I designed an additional method, `group_detections_by_entity`, which reorganizes these results by entity rather than by frame. This method maps each entity ID to its corresponding observations across frames, allowing structured access to a given object's trajectory.

In addition, I implemented an `Observation` class. Each instance of this class encapsulates the details of a single detection, including bounding box coordinates, class ID, confidence score, and the frame index.

For readability, the complete implementation code of these methods is provided on GitHub. The link is provided in the Appendix.

2.3 Person Segmentation

Thanks to our dedicated work on the creation of a proper dataset and the subsequent training of the YOLO model on this specific dataset, we've been able to obtain a highly performing model for analyzing such difficult videos.

We implemented the YOLO model as one of the first, yet most important, steps in our pipeline. Indeed, with this integration, we can now precisely obtain all bounding boxes for a specific person throughout the video. By ensuring the high quality and robustness of these initial bounding box detections, we lay the essential groundwork for achieving significantly better and more reliable segmentation masks in subsequent stages, thereby substantially enhancing our system's overall performance.

2.3.1 Presentation of Segment Anything model

Developing a custom segmentation model for videos is extremely complex and computationally expensive. It typically requires a large, annotated dataset (although some public datasets do exist), and the training process itself demands significant GPU resources and time.

To circumvent these challenges, I chose to use a state-of-the-art, ready-to-use segmentation model: the **Segment Anything Model (SAM)** [8], developed by Meta (formerly Facebook). SAM is one of the most powerful and general-purpose segmentation models currently available.

SAM is a promptable segmentation system. It means, there are several ways to interact with SAM for image or video segmentation:

- **Prompt-based segmentation:** You provide the model with a textual or spatial prompt to guide the segmentation.
- **Point-based interaction:** You manually click on the object to indicate its location; SAM then returns the corresponding segmentation mask.
- **Bounding box-based prompts:** You provide the coordinates of a bounding box enclosing the object, and the model returns the associated segmentation mask. This is the method I used, as it integrates easily with a pre-existing object detection pipeline.

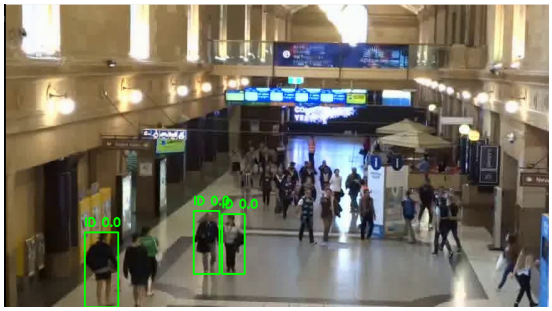
An additional functionality available through Ultralytics' implementation of SAM is **mask propagation**, which enables the segmentation of an object across multiple video frames based on its initial bounding box or mask. However, this process is computationally very expensive. On my setup (CPU only), propagating masks over just 5 frames took approximately 25 minutes. Due to this limitation, I opted not to use propagation. My object detection-based pipeline remains relevant and efficient in this context.

2.3.2 Integration into the Pipeline

After applying the object detection model to the video and obtaining the bounding boxes for all individuals, we use predefined person IDs to select specific people of interest. Subsequently, SAM is applied only to those selected individuals in each frame.

A key consideration lies in the fundamental difference between how YOLO and SAM operate, which impacts their integration strategy. Two main approaches were tested:

- **Initial approach:** The video is first preprocessed by extracting and resizing the frames. Then, for each frame, YOLO is applied to detect persons, followed by SAM to extract a segmentation mask for the selected person IDs. While functionally correct, this approach was extremely computationally expensive, requiring approximately 15 seconds per frame.
- **Optimized approach:** The video is still preprocessed by extracting and resizing the frames to 864×484 pixels. However, instead of applying YOLO on each frame individually, the object detection is now performed over the entire video in a single pass using the highly optimized implementation provided by Ultralytics. This results in a significant speed-up, reducing object detection time to 50 milliseconds per frame. Once all detections are collected, SAM is applied on the relevant frames and person IDs, now requiring approximately 2 seconds per frame for segmentation. This leads to a considerable overall improvement in pipeline efficiency.



(a) Visualizing YOLO detection and bounding boxes

(b) Corresponding segmentation mask computed by SAM

Figure 2.16: Example of computing a segmentation mask based on bounding boxes detected by YOLO

Figure 2.16 illustrates the segmentation pipeline in action. After applying the custom YOLO model and selecting specific person IDs, bounding boxes are overlaid on the original frame. These boxes are then used as prompts for SAM, which generates the corresponding segmentation masks. These masks are saved in a specific folder to be then used for the next step. This visual output is essential for validating both the accuracy of the detections and the effectiveness of the segmentation process.

Regarding the concrete code implementation in the pipeline, the masking module is built around the `Masker` class and its `mask` method, which generates a binary mask from a list of `Observation` instances. The main challenge in this function was understanding how to access the mask produced by SAM.

In practice, the workflow consists of converting several boolean masks into NumPy boolean arrays, and then combining them into a single integer binary mask (with values 0 or 255). Finally, this aggregated mask is saved to a designated folder for downstream processing.

2.3.3 Evaluation of the Segmentation

Evaluating segmentation models typically relies on standard metrics such as Intersection over Union (IoU), Dice coefficient, or pixel-wise accuracy. However, all these metrics require access to ground truth segmentation masks, which are usually created manually.

Given the time constraints and the absence of annotated data for my specific use case, I chose to conduct a qualitative evaluation of SAM's outputs. This involved visually inspecting the predicted masks frame by frame to assess whether the segmented regions correctly matched the targeted objects (in this case, specific people identified by the object detection step).

Although this approach does not provide objective performance scores, it still offers valuable insights into the reliability of the segmentation, particularly in identifying failures or inconsistencies across frames.

2.4 Inpainting Module

The inpainting module represents the final and most challenging step of our pipeline. The goal is to use the segmentation masks generated by SAM and the associated frames to remove selected individuals from a video while reconstructing the background in a visually coherent way.

We feed the masks as well as the frames into a video inpainting model, which attempts to fill in the masked regions across consecutive frames. However, several major challenges emerged during this phase:

- **Trade-off between image resolution and performance:** I encountered a classic trade-off situation: higher-resolution frames tend to yield better inpainting quality, but they also require significantly more memory. When using large images, my machine frequently crashed due to RAM saturation. I had to downscale the frames to 864×484, which was the maximum resolution my system could handle without failure. Unfortunately, this resolution often limited the quality of the inpainting results.
- **Computational cost:** The inpainting process is highly resource-intensive. Processing a short 5-second video (125 frames at 25 FPS) took approximately 20 minutes. This high latency made iterative experimentation slow and frustrating, particularly without access to GPU acceleration.

This section presents the different inpainting models I experimented with and provides an analysis of the results obtained from each approach.

2.4.1 End-to-End Framework for Flow-Guided Video Inpainting

The **End-to-End Framework for Flow-Guided Video Inpainting (E2FGVI)** [9] is the first model I tried because the performance of the model on the paper sounds just extroardinary. The results of the model can be seen on the Github of the project [10]. Their paper was accepted to the Computer Vision and Pattern Recognition Conference (CVPR) in 2022. The acceptance of a paper in this conference is always a quality guarantee.

Limitations of Prior Flow-Based Video Inpainting Methods

Traditional flow-based methods treat video inpainting as a pixel propagation problem, aiming to preserve temporal coherence. These approaches typically consist of three interdependent stages:

- **Flow Completion:** Estimate and complete optical flow in corrupted regions, since its absence compromises later processes.
- **Pixel Propagation:** Fill missing areas by propagating pixels from visible regions, guided by the completed flow.
- **Content Hallucination:** Use a pre-trained image inpainting network to hallucinate content in regions that cannot be recovered by propagation.

While these methods have shown promising results, they suffer from several key limitations due to their hand-crafted and disjointed design:

- **Error Accumulation and Amplification:** Errors from earlier stages (e.g., inaccurate flow) propagate and worsen in later stages. For instance, incorrect flow can misguide pixel propagation and confuse the hallucination stage.
- **High Computational Cost:** Many operations (e.g., Poisson blending, sparse linear solvers, per-pixel indexing) are not GPU-friendly. Some methods (like DFVI) take up to 4 minutes to inpaint a 70-frame video, making them impractical for real-world deployment.
- **Lack of Temporal Coherence in Hallucination:** Relying solely on image-level inpainting disregards temporal dependencies, often resulting in temporally inconsistent outputs.

E2FGVI's Approach to Address These Limitations

To overcome these issues, **E2FGVI** (End-to-End framework for Flow-Guided Video Inpainting) introduces a fully trainable pipeline composed of three modules, inspired by the traditional stages but optimized jointly:

- **Joint Optimization:** Flow completion, feature propagation, and content hallucination are optimized together. This end-to-end training avoids the error compounding of modular pipelines and ensures efficiency and accuracy.
- **One-Step Flow Completion:** E2FGVI uses a single-step flow completion directly on masked videos ($<$0.01s/flow), which is significantly faster than prior multi-stage flow estimation ($>$0.4s/flow).
- **Feature-Level Propagation via Deformable Convolutions:** Instead of propagating pixels, E2FGVI performs flow-guided propagation in the feature space. Deformable convolutions allow learnable offsets, making the system more robust to flow inaccuracies and more computationally efficient.
- **Temporal Focal Transformer:** For content hallucination, E2FGVI introduces a transformer module that models both spatial and long-range temporal dependencies (local and non-local). This ensures strong temporal coherence in the inpainted frames.
- **Performance and Efficiency:** E2FGVI outperforms prior methods both qualitatively and quantitatively, processing 432×240 resolution videos at 0.12 seconds per frame—making it nearly $15\times$ faster than traditional flow-based approaches.

2.4.2 Results with E2FGVI

First experiments with E2FGVI

My initial experiments with the E2FGVI model were somewhat disappointing. Based on the qualitative results presented in the official GitHub repository [10], I expected much better performance. Figure 2.17 shows selected frames extracted from the demonstration video posted on the repository. This video is notably challenging, as it depicts a scene from the movie *Spider-Man* where the protagonist flies from building to building, making any artifacts or imperfections in the inpainting highly visible.

(a) Frame 1

(b) Frame 3

(c) Frame 6

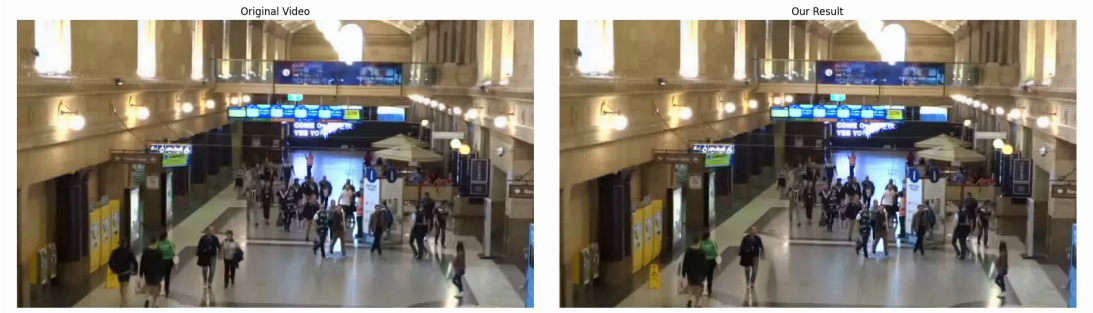
Figure 2.17: Selected frames from the original E2FGVI video inpainting results (frames 1, 3, and 6)

When applying the model to my own videos, the results were considerably less satisfactory. I observed a flickering phenomenon where the persons I aimed to remove appeared intermittently and inconsistently. Ultimately, the subjects remained partially visible, leading to unsatisfactory results. Due to the nature of this issue, illustrating it through static images is ineffective; the flickering is best observed in video format, which is not included here.

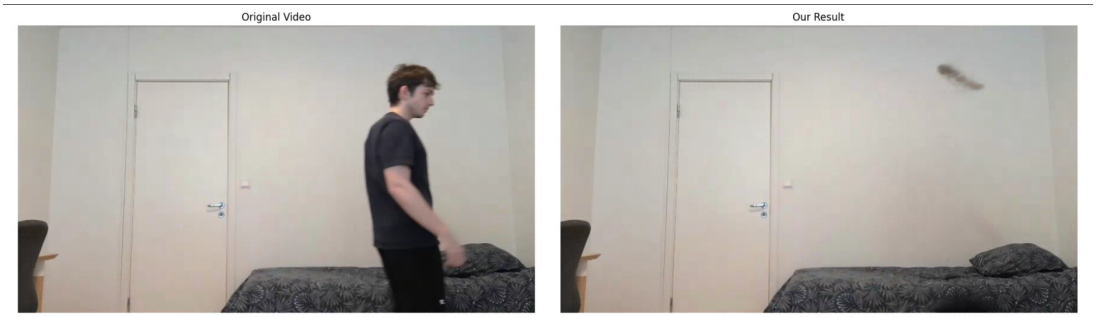
Final results

After discussing these initial results with my supervisor, he identified a crucial problem related to video preprocessing. To fit the model's input requirements, I had downscaled my videos to 640×640 pixels, corresponding to the YOLO detection model's input size, and applied this resolution consistently throughout the pipeline. However, this resizing distorted the original aspect ratios, leading to unrealistic object proportions and negatively impacting the model's performance.

Following his advice, I adjusted the resizing strategy to 864×480 pixels, which better preserved the original aspect ratio of the videos. This change significantly improved the results, enabling the model to produce qualitatively acceptable inpainting outputs. This was a classic pitfall related to improper preprocessing, from which I learned an important lesson.



(a) situation 1



(b) situation 2

Figure 2.18: Selected frames from the videos tested after adjusting preprocessing

Evaluating video inpainting quality from static frames alone remains challenging, as temporal artifacts like flickering are difficult to capture. Overall, the results obtained with E2FGVI after correction are quite promising. However, I could not replicate the same quality as showcased in the official teaser, even on relatively simple videos such as the second example shown in Fig. 2.18. This raises questions about the model's robustness and generalization capability in practical settings.

2.4.3 Integration into the Pipeline

The integration in the pipeline is quite simple and relies on a single line of code using the Python module subprocess to execute the model :

```
1     if "inpainting" in STEPS:
2         result = subprocess.run(
3         f"python {INPAINTING_INFERENCE_PATH} --model e2fgvi_hq --video {
    DATA_PATH_PROCESSED}/{PROJECT_NAME}/frames --mask {OUTPUTS_PATH}/{
    PROJECT_NAME}/masks --ckpt {INPAINTING_MODEL_PATH} --set_size --width
     864 --height 480", shell=True)
4
```

Listing 2.1: Inpainting module in the pipeline

2.4.4 Diffusion Models

Diffusion models are a class of generative models that have recently achieved state-of-the-art results in image synthesis, editing, and inpainting tasks. That's why my supervisor recommended me to try them. Their core idea is inspired by non-equilibrium thermodynamics: they learn to reverse a gradual noising process applied to data.

The process consists of two main phases:

- **Forward (diffusion) process:** Starting from a clean image, random Gaussian noise is progressively added over several time steps until the image becomes nearly pure noise. This process is fixed and non-trainable.
- **Reverse (denoising) process:** A neural network, typically a U-Net, is trained to learn the reverse of the diffusion process i.e., starting from noise, it reconstructs the original image step-by-step. During training, the model learns to predict and remove noise at each timestep.

In the context of video or image inpainting, the model is conditioned not only on the noisy image but also on auxiliary information such as a mask or a prompt. This conditioning guides the generation to reconstruct only the missing or masked parts while keeping the rest unchanged.

Diffusion models are known for producing high-fidelity results in image generation tasks. However, they are computationally expensive due to the large number of iterative denoising steps required. Furthermore, most available diffusion-based tools focus primarily on image inpainting, without modeling temporal consistency, an essential requirement for video inpainting. For these reasons, diffusion models were not the first approach I explored.

Nonetheless, given their popularity in recent state-of-the-art generative tasks (image synthesis, inpainting, and even video generation), I was advised by my supervisor to experiment with them.

Experimenting with Diffusion Models

To avoid spending excessive time installing and integrating complex repositories without guarantee of effectiveness, I opted for a quick prototyping solution. I used the online platform **Clipdrop** [11]. Clipdrop is a commercial tool developed by Init ML, backed by Stability AI (creators of Stable Diffusion), designed for AI-based editing tasks such as background removal, relighting, upscaling, and inpainting. Specifically, I tested their *Cleanup* tool, which allows object removal from images using a mask and then reconstructs the background using a diffusion-based model.

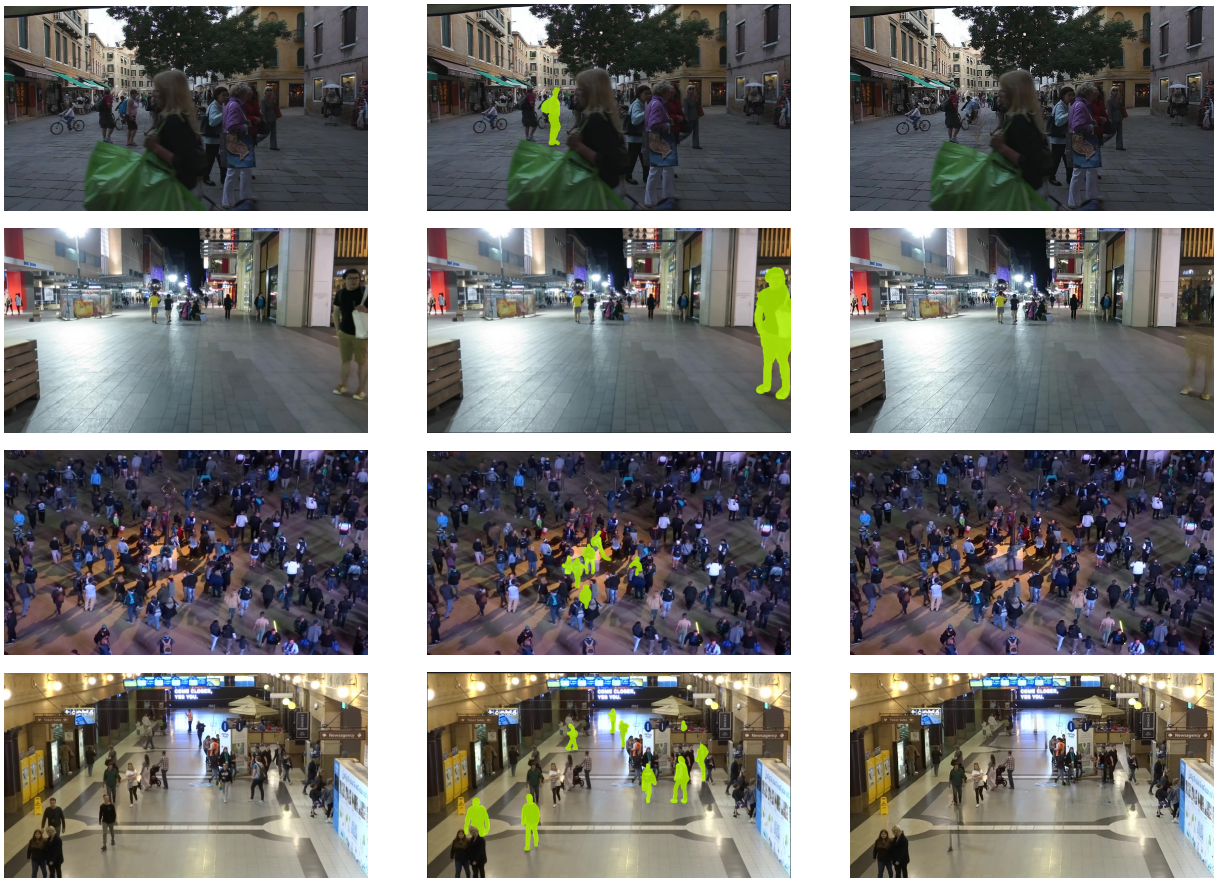


Figure 2.19: Qualitative results of diffusion-based inpainting using Clipdrop. From left to right: original image, user-provided mask, and inpainted output.

Figure 2.19 illustrates the results of these experiments where each row represents a test case. The first column shows the original image, the second column displays the user-defined mask (i.e., the area to be removed), and the third column shows the generated inpainted result.

2.4.5 Results with Diffusion models

Overall, the results were not convincing. In many cases, residual artifacts or semi-transparent shapes remained visible, especially when the masked object was surrounded by several others. For example, the second result on the second row shows clear artifacts and unrealistic blending.

Figure 2.20 shows a close-up of one such failure case. The inpainted region lacks texture consistency and structural coherence with the surrounding area.

Figure 2.20: Zoomed-in view of the fourth inpainting test.

These models may perform adequately on simple images with uniform backgrounds and few elements. However, the scenes in my dataset (MOT17/MOT20) are challenging: crowded, cluttered, and dynamic. Moreover, diffusion models typically do not enforce temporal coherence, which is critical in video applications. Even when individual frame results are visually acceptable, applying the model frame-by-frame introduces significant inconsistencies when reconstructing the full video sequence. This temporal instability renders the approach unsuitable for real-world video inpainting tasks in its current form.

2.5 Implementation Details

This section provides technical insights into the development environment and project organization that supported the implementation of the video processing pipeline. Each section responds to a problem I faced and provide the solution I implemented.

2.5.1 Development Environment

Development under Linux

For this project, I chose to work under a Linux environment due to its stability, performance, and the superior developer experience it offers, particularly for computer vision and deep learning tasks. Initially, I used a virtual machine (VM) hosted on Windows. However, I soon faced concerns about performance limitations and the potential instability of the VM. To address this, I set up a dual-boot configuration with Ubuntu, which was also the preferred operating system of most researchers in the lab. Although the setup was slightly stressful and technically challenging, it significantly improved my system-level knowledge and provided a more robust working environment.

Virtual Environments

During the early weeks of my internship, one of the researchers assisted me in setting up a Python virtual environment for a project on Windows. At that time, I was unfamiliar with virtual environments, as they were rarely required in academic projects. Over time, I came to appreciate the importance of isolating project dependencies and managing Python versions effectively.

As soon as I began working on the video processing pipeline, setting up a dedicated virtual environment became my first step. I used `pyenv` to manage multiple Python versions and `virtualenv` to isolate the project environment. This approach helped avoid dependency conflicts and ensured reproducibility across development sessions.

Version Control

My initial experience with version control was similar to that of virtual environments. A researcher helped me configure access to the lab's GitLab repository using SSH keys. However, due to my limited experience with Git at that time, I was unable to use it effectively.

When I switched to Linux, I took the initiative to set up my own SSH access to GitHub and dedicated time to learning Git fundamentals through online tutorials, including concepts such as branching, merging, and rebasing. I then used Git and GitHub throughout the implementation of the video pipeline project. Today, I consider Git and Python virtual environments as essential tools for managing any professional software project efficiently and reliably.

2.5.2 Project Architecture

As different models were tested to obtain a functional pipeline, the project structure quickly became messy. After a few weeks, I decided to refactor the project architecture to improve clarity and maintainability.

I also chose to use object-oriented programming, creating dedicated classes to manage the different steps of the pipeline. This modular approach significantly improved the scalability and reusability of the codebase.

Below is a brief description of the main folders shown in Fig.2.21:

- **configs**: Contains the various JSON configuration files used to convert the initial datasets into the YOLO format, along with tracker configuration settings.
- **dataset**: Includes a `raw` subfolder with unprocessed MOT datasets, and a `processed` subfolder containing resized frames and videos prepared for model evaluation.
- **models**: Contains the different models used in the pipeline, such as YOLO, object masking, and video inpainting models. Note: masking and inpainting models are not shown in the figure due to the large number of associated files.
- **modules**: Includes Python classes for different stages of the pipeline. It also includes a custom `utils` module, which manages files, geometry transformations (e.g., YOLO format conversion), image, and video functions. The `data` submodule handles object detection and post-tracking formatting.
- **outputs**: Used to save intermediate and final results generated by pipeline modules during experimentation.
- **scripts**: Contains utility scripts such as dataset conversion routines.
- **tests**: Contains integration tests for various pipeline stages (e.g., detection, inpainting).
- **training**: Stores YOLO-formatted datasets and training configuration files.

An important file in the project is `pipeline.py`, which is responsible for orchestrating the different modules and scripts to produce a fully functional video processing pipeline. It serves as the main entry point of the project.

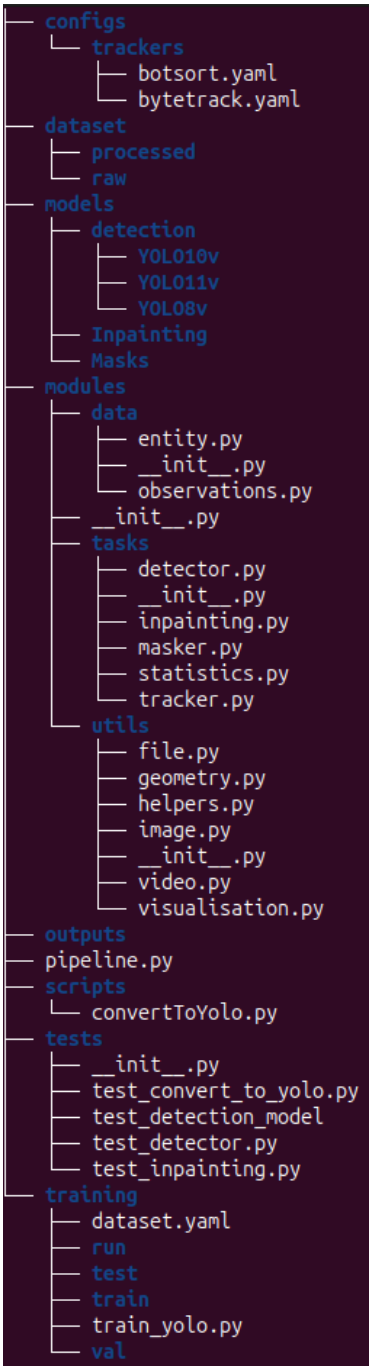


Figure 2.21: General architecture of the project showing the main folders and modules involved in the pipeline.

Training on a Remote Server

During the training phase of the YOLO model, I initially used my personal computer. However, I quickly noticed that it consumed a large amount of resources, leading to latency and reduced performance during regular use. To address this, I took advantage of the fact that someone in my family had rented a Virtual Private Server (VPS) and agreed to grant me access.

One interesting aspect was the very first question he asked: which software and Python packages would be required for my project. This was to avoid potential conflicts with the dependencies used in his own work. That discussion made me realize, once again, the importance and flexibility of working with isolated Python environments. Thanks to `pyenv` and `virtualenv`, I was able to clearly explain that my environment would be entirely isolated and would not interfere with any existing setup. This experience further reinforced my understanding of environment management and its importance in collaborative and multi-user systems.

2.6 Conclusion & Future Work

2.6.1 Summary of Achievements

Throughout this project, several technical milestones were reached:

- Development of a robust Python script to convert raw datasets into YOLO-compatible format.
- Successful training of the YOLO model on a custom dataset and thorough analysis of the results.
- Design and implementation of a complete video processing pipeline integrating object detection, segmentation, and video inpainting modules.

2.6.2 Personal Learning Outcomes

Beyond technical knowledge in Computer Vision specifically object detection, segmentation, and inpainting. This project allowed me to acquire skills that are essential for real-world engineering projects:

- Understanding and implementing data preprocessing pipelines (Data Engineering).
- Managing a project using virtual environments and version control tools (MLOps fundamentals).
- Structuring a complex project for modularity and scalability using object-oriented programming.

The biggest challenge was ensuring that all components worked harmoniously together. This required both technical rigor and methodological thinking.

2.6.3 Limitations of the Current System

One limitation lies in the inpainting module. Due to limited hardware (particularly GPU resources), I had to reduce image resolutions significantly, which compromised quality and made experimentation difficult. This trade-off between feasibility and performance was a recurring challenge.

Another limitation is usability: currently, the pipeline is fully script-based. A potential improvement would be the development of a lightweight web platform allowing users to upload videos and run the pipeline through a user interface. This would make the tool more accessible and impactful beyond a research context.

2.6.4 Future Work

Future improvements could include:

- Exploring more advanced inpainting models that are optimized for performance on lower-end hardware.
- Packaging the project as a deployable service (e.g., via a Docker container or Flask API).
- Creating a user-friendly front-end to facilitate broader adoption of the pipeline.

3. 3D Reconstruction: Structure from Motion Pipeline

From early April to late May, in parallel with my main project on video inpainting, I had the opportunity to attend an advanced course in 3D Computer Vision, taught by Per-Erik Forssén and Mårten Wadenbäck, both researchers and lecturers at the Computer Vision Laboratory (CVL). The course combined a rigorous mathematical foundation with hands-on implementation of algorithms for inferring 3D structure from images.

The main objective of the course was to implement a complete Structure from Motion (SfM) pipeline. I collaborated with three PhD students, two from CVL and one from another department. The project culminated in an oral presentation with a live demonstration of our pipeline, as well as the submission of a detailed technical report documenting our methodology, implementation, and results.

3.1 Problem Overview and Terminology

As illustrated in Figure 3.1, the goal of Structure from Motion (SfM) is to reconstruct a 3D model of an object or scene from a set of 2D images taken from different viewpoints. This problem lies at the intersection of computer vision and projective geometry and requires the integration of several sophisticated components, such as feature detection, pose estimation, triangulation, and bundle adjustment.

A key challenge in SfM is the accurate estimation of camera poses i.e., their positions and orientations as it directly impacts the quality and consistency of the reconstructed 3D point cloud.

3.1.1 Applications

Structure from Motion is widely used in fields such as robotics, autonomous navigation, augmented reality, heritage digitization, and cartography. Notably, researchers at Linköping University have contributed to developments in this domain that have been integrated into large-scale systems such as Google Earth, demonstrating the real-world impact of academic research in SfM.

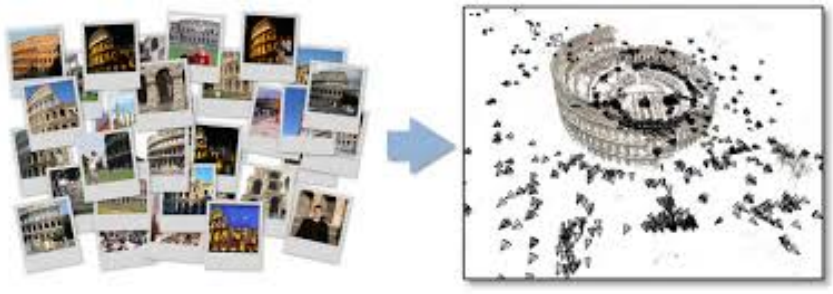


Figure 3.1: 3D reconstruction of the Colosseum from multiple images, highlighting the estimated camera poses used during the Structure from Motion pipeline.

3.2 Structure of the Course

The course was structured around both lectures and lab sessions (CE1 and CE2). It focused on providing the mathematical foundations required to address 3D reconstruction problems, as well as presenting the core algorithms used to build practical pipelines by leveraging these mathematical tools.

Table 3.1: Lecture and Lab Schedule

Session	**Content**
Lecture 1	Introduction, 3D perception, and local features
Lecture 2	Maximum likelihood estimation, RANSAC, and the essential matrix
Lecture 3	Non-linear least squares estimation and robust error norms
CE1 (Lab)	Non-linear optimization and the gold standard algorithm
Lecture 4	Multi-view stereo, correspondence fields, and triangulation
Lecture 5	Representations of 3D rotations
Lecture 6	Absolute and relative camera pose, and minimal cases
CE2 (Lab)	Densification using PatchMatch
Lecture 7	Structure from motion, bundle adjustment, and project kickoff
Guest Lecture 1	*(Not attended)*
Guest Lecture 2	Maxar: Technical overview of satellite imagery and 3D reconstruction
Final Seminar	Project presentations, exam overview, and wrap-up session

As shown in Fig. 3.1, the course covered a broad range of topics, which was one of the main challenges. Each lecture introduced new algorithms contributing to a complete 3D reconstruction pipeline. I initially struggled to understand how each component fit into the pipeline, its role, inputs, outputs, and interactions with others.

Some algorithms, like **Random Sample Consensus (RANSAC)** [12], acted as robustness layers over others, adding an extra level of abstraction. A further difficulty was my limited background in certain mathematical areas, especially projective geometry.

Over time, I realized that mastering every theory in depth was less critical than understanding how the algorithms work together within the pipeline.

The course included two lab sessions requiring us to implement several core components. I worked closely with a PhD student to tackle these exercises. The labs were only

loosely guided, demanding significant preparation and independent problem-solving.

Although the expected time per lab was 4–6 hours, we often spent 3–4 times more to grasp the concepts and solve the tasks. Each lab was followed by a 4-hour supervised session, mostly aimed at validating our results to earn course credit.

While I wasn't required to earn credits, my partner was, which motivated me to invest fully in the labs for both his success and the robustness of our final project. We had two weeks after Lecture 7 to complete the final submission.

3.3 Multi-View 3D Reconstruction Pipeline (Simplified Overview)

This section outlines the key steps in our 3D reconstruction pipeline, using concepts defined earlier. The process takes multiple calibrated images as input and outputs a dense 3D point cloud and camera poses.

1. **Feature Detection and Description:** In each image, we detect interest regions (e.g., corners, blobs, edges). These regions are referred to as **keypoints**. To describe the local appearance around each keypoint, we compute a **feature descriptor**, which is a vector encoding distinctive visual properties, enabling robust matching across images.

2. **Feature Matching:** Using the feature descriptors, we identify correspondences between keypoints in consecutive image pairs (i.e., between image $i-1$ and image i). Matching is performed using nearest neighbor search, and ambiguous matches are filtered using methods such as the Lowe ratio test and RANSAC. The result is a set of reliable point correspondences between images.

3. **Estimation of Geometric Relations:** From the set of correspondences, we estimate two key matrices:

   - The **Fundamental Matrix (F)**: encapsulates the epipolar geometry between two views in pixel coordinates.
   - The **Essential Matrix (E)**: derived from F using camera intrinsics, it enables recovering the relative pose (rotation and translation) between the two camera positions.

   These estimations are carried out using robust algorithms such as the **normalized 8-point algorithm** [13] or RANSAC.

4. **Triangulation:** Once the relative poses of two views are known, we compute the 3D coordinates of matched points by intersecting the corresponding projection rays in 3D space. This results in a first **sparse point cloud** representing the scene structure.

5. **Camera Pose Estimation (Perspective-n-Point, PnP):** As new images are added, we estimate their position and orientation (pose) by solving the PnP problem. This involves finding the camera pose that best explains the projection of known 3D points onto the new image. We use a P3P solver within a custom RANSAC loop to ensure robustness to outliers.

6. **Bundle Adjustment:** We perform a global non-linear optimization that jointly refines all estimated 3D points and camera poses. This step minimizes the overall reprojection error, significantly improving the accuracy and consistency of the reconstruction.

7. **Re-triangulation:** After bundle adjustment, we recompute the 3D positions of the points using the refined camera poses. This step improves accuracy by replacing the initial linear triangulation with a more geometrically consistent estimate.

8. **Dense Stereo Matching:** Finally, we densify the sparse point cloud using a stereo matching algorithm (**PatchMatch** [14]). This step estimates a depth value for nearly every pixel, producing a dense 3D reconstruction of the scene.

3.4 Contribution to the pipeline

3.4.1 Feature Extraction

My main contribution to the pipeline was on the feature extraction component, which encompasses three key steps: feature detection, feature description, and feature matching.

Feature Detection and Description

For feature detection and description, I used the **Scale-Invariant Feature Transform (SIFT)** [15] algorithm, a well-known for its robustness to scale, rotation, and illumination changes.

SIFT operates by constructing a *scale space*, a set of progressively blurred images in which keypoints are detected as local extrema in the **Difference of Gaussians (DoG)**. This allows the detection of features that remain stable across different image scales.

Each keypoint is then assigned an orientation based on local image gradients, and a feature descriptor is computed by sampling gradient orientations around the keypoint. The resulting descriptors are highly distinctive and invariant to common image transformations, making them suitable for reliable matching across multiple views.

Feature Matching

To match descriptors between image pairs, I combined the following two approaches:

- **Brute-Force Matching (BFM):** For each descriptor in the first image, the Euclidean distance is computed to all descriptors in the second image. The match with the smallest distance is selected.

- **K-Nearest Neighbors (k-NN) with Lowe's Ratio Test:** To improve robustness and reduce false matches, I used the 2-nearest neighbors (2-NN) strategy combined with Lowe's ratio test. A match is retained only if the ratio between the distances to the best and second-best matches is below a threshold (typically 0.75), ensuring that the best match is significantly more reliable.

Despite these improvements, many initial correspondences may still be incorrect due to noise, occlusions, or repetitive textures. To further filter out outliers, I applied the **Random Sample Consensus (RANSAC)** algorithm.

- RANSAC iteratively selects random subsets of matches to estimate a geometric model. In this case, the *Fundamental Matrix*.
- It then determines how many of the total matches are consistent with this model (inliers).
- The model with the highest number of inliers is selected, and only the corresponding matches are retained for the following stages.

This step is critical to ensure that only geometrically consistent correspondences are used in the subsequent pose estimation and triangulation stages. We had previously studied and implemented this procedure in the CE1 assignment, which significantly facilitated its integration into the full SfM pipeline.

3.4.2 Re-triangulation

I contributed to the **re-triangulation** step, which aimed to refine 3D point estimates using updated camera poses obtained after bundle adjustment. Our initial objective was to implement full multi-view triangulation, which aggregates observations from all available views to improve accuracy. However, this required substantial refactoring of tightly coupled data structures in the codebase, which proved infeasible within the project's time constraints.

As an alternative, we explored optimal two-view re-triangulation. However, this approach introduced numerical instability and degraded key evaluation metrics such as reprojection error and point cloud consistency.

We ultimately adopted a pragmatic compromise: selectively re-triangulating only the 3D points with high reprojection errors. This local refinement had limited impact on overall reconstruction quality, but it allowed us to mitigate some of the worst geometric inconsistencies without destabilizing the pipeline.

Several factors contributed to the limited effectiveness of this method:

- Absence of filtering for poorly observed or outlier points,
- Limited multi-view visibility for most 3D points,
- Local overfitting effects introduced by bundle adjustment.

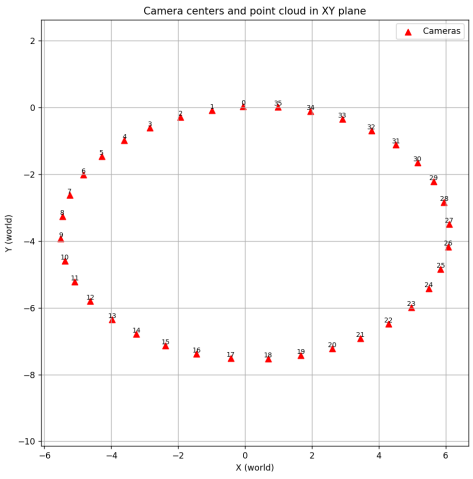
For more robust and consistent results, a proper multi-view triangulation strategy would be necessary—one that fuses observations across all available camera views to re-estimate each 3D point with higher geometric precision.

3.5 Example of Result

We tested our pipeline on several datasets, starting with the *Dinosaur* dataset, which includes 36 images captured in a controlled environment using a rotating camera around a static model. An example frame is shown in Fig. 3.2.

Figure 3.2: Frame 17 from the Dinosaur dataset

Our Structure from Motion pipeline successfully estimated both the camera trajectory and the 3D scene geometry, as illustrated in Fig. 3.3.



(a) Estimated camera positions

(b) Recovered 3D point cloud (20,386 points)

Figure 3.3: Reconstruction results from the Dinosaur dataset

Conclusion

This internship at the Computer Vision Lab (CVL) marked a significant step in my journey into the field of Computer Vision. I had the opportunity to explore various domains, from object detection and tracking to structure-from-motion and 3D reconstruction.

In the first project, I contributed to building a robust video-processing pipeline. My main task focused on data engineering: I developed a script to convert custom datasets into the YOLO format, enabling the training of a specialized object detection model for people. The improved model performs reliably even in challenging conditions, such as low lighting or dense crowds—capabilities that were previously lacking. I also worked on generating precise masks by integrating object detection, multi-object tracking, and segmentation, laying the groundwork for further tasks like video inpainting, which remains a resource-intensive challenge.

In the second part, developing a full structure-from-motion pipeline allowed me to gain a deeper understanding of complex concepts such as projective geometry, feature matching, and bundle adjustment. It also gave me a broader perspective on how various computer vision algorithms interact within a complete and functional system.

Overall, this experience strengthened both my technical skills and my ability to work independently on open-ended problems. It confirmed my strong interest in Computer Vision and further motivated me to pursue advanced work in this field

Bibliography

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *arXiv preprint arXiv:1311.2524*, 2014.

[2] N. Aharon, R. Orfaig, and B.-Z. Bobrovsky, "Bot-sort: Robust associations multi-pedestrian tracking," *arXiv preprint arXiv:2206.14651*, 2022.

[3] Y. Zhang, P. Sun, Y. Jiang, D. Yu, F. Weng, Z. Yuan, P. Luo, W. Liu, and X. Wang, "Bytetrack: Multi-object tracking by associating every detection box," *arXiv preprint arXiv:2110.06864*, 2022.

[4] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," *arXiv preprint arXiv:1703.07402*, 2017.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *arXiv preprint arXiv:1506.02640*, 2016.

[6] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," *arXiv preprint arXiv:1405.0312*, 2015.

[7] A. Milan, L. Leal-Taixe, I. Reid, S. Roth, and K. Schindler, "Mot16: A benchmark for multi-object tracking," *arXiv preprint arXiv:1603.00831*, 2016.

[8] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, "Segment anything," 2023.

[9] Z. Li, C.-Z. Lu, J. Qin, C.-L. Guo, and M.-M. Cheng, "Towards an end-to-end framework for flow-guided video inpainting," *arXiv preprint arXiv:2204.02663*, 2022.

[10] Z. Li, C.-Z. Lu, J. Qin, C.-L. Guo, and M.-M. Cheng, "E2fgvi." `https://github.com/MCG-NKU/E2FGVI`, 2022. Accessed: 2025-07-30.

[11] InitML, "Clipdrop." `https://clipdrop.co/`, 2025. Accessed: 2025-07-30.

[12] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, 1981.

[13] H. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections," *Nature*, 1981.

[14] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "Patchmatch: A randomized correspondence algorithm for structural image editing," in *ACM Transactions on Graphics (ToG)*, vol. 28, p. 24, ACM, 2009.

[15] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

Appendix

The full implementation, including source code and additional resources, is available on **GitHub**: *github.com/MathisPicasse/YOLOV11_SAM2_E2FGVI*