SAPIENZA
UNIVERSITÀ DI ROMA

Mixed Reality for Industry 4.0: A case study of a support for the operator

Facoltà di Ingegneria dell'informazione, informatica e statistica
Corso di Laurea Magistrale in Computer Science

Candidate
Daniele Bertagnoli
ID number 1903768

Thesis Advisor
Prof. Luigi Cinque

Co-Advisor
Prof. Marco Raoul Marini

Academic Year 2023/2024

Thesis not yet defended

Mixed Reality for Industry 4.0: A case study of a support for the operator
Master's thesis. Sapienza – University of Rome

© 2024 Daniele Bertagnoli. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: bertagnoli.1903768@studenti.uniroma1.it

"No. Try not. Do... or do not. There is no try."

$\sim$ *Master Yoda*

Ringraziamenti

Voglio innanzitutto ringraziare la società Thales Alenia Space per avermi dato la possibilità di realizzare questo progetto. Un sentito grazie va anche al mio relatore, il professor Luigi Cinque, e al co-relatore, il professor Marco Raul Marini, per avermi seguito durante tutto lo svolgimento del progetto, sia all'interno che all'esterno dell'azienda, facendomi entrare a far parte, anche se brevemente, del gruppo di ricerca del VisionLab.

Ci tengo a specificare che questi ringraziamenti non seguiranno un ordine di preferenza; andrò a memoria, sperando di non dimenticare nessuno!

Le prime persone che ci tengo a ringraziare sono i miei genitori, che oltre ad aver finanziato tutto il mio percorso di studi (rinunciando ad almeno una Ferrari, come dice sempre papà), mi hanno sempre supportato e sopportato in questi cinque anni tra triennale e magistrale. Grazie per le torte a fine esame, anche se forse papà era quello che le apprezzava di più, non so se perché era contento di vedermi felice o perché finalmente poteva mangiare qualche dolce. Grazie anche a mia sorella Nicole, che pur non essendo in casa mi ha sempre chiesto come stessero andando le cose (anche perché viene a mangiare da noi un giorno sì e l'altro pure). Ovviamente, grazie anche a mio cognato Andrea, che ha dispensato consigli utili su come affrontare i corsi. Menzione d'onore per Choco che quando non c'era nessuno a casa veniva automaticamente sequestrato per sentirmi ripetere prima di un esame, se solo potessi parlare so che sapresti dimostrare interamente almeno due enunciati sui network graphs.

Grazie anche ai miei quattro nonni. Nonno Domenico e nonna Lina, finalmente, quando mi chiederete 'Ma quanto ti manca?', potrò rispondere che ho finito e che posso iniziare a lavorare per portare il pane a casa. Nonno Giuseppe e nonna Teresa, anche se non siete qui fisicamente per festeggiare, so che lo farete comunque, guardandomi da dove siete ora.

Voglio anche ringraziare i miei amici e ormai ex-colleghi universitari. Giacomo, per gli amici Jack, con cui ho condiviso pomeriggi e nottate lavorando ai vari progetti, discutendo di quanto Linux fosse superiore a Mac o Windows, se fosse più conveniente il tiling rispetto al floating window management o se avesse senso cambiare distro. Giuseppe, che ha dimostrato di essere un grande capitano, capace di governare la ciurma nonostante un mozzo ubriacone sulla nave. E infine Marco, con cui ho trascorso ore e ore all'interno dello Space Lab 4.0 di Thales Alenia Space, e che ha reso tutto più leggero, tra una chiacchiera sulla palestra e una sui meme politicamente scorretti.

Ora è il turno di ringraziare gli amici di una vita, il Triplo Marco. Non ci sono parole per descrivere quanto sia stato importante avere così tante persone al mio fianco. Le chiacchierate davanti a un'ottima Coca-Cola Zero da Paride o da Fat-Max, i meme, le partite al Pro Club, i calcetti e le vacanze sono ricordi che porterò sempre con me. Ci tengo a ringraziarvi uno a uno, sperando di non dimenticare nessuno: Alessio, Anja, Cecco, CG, Chris, Chiara, Disa, Guaro, K, Khodja, Miriam, Mirko,

Matteo, Martina, Nicchio, Verans e Veronica. Grazie ancora per tutto quello che avete fatto, anche se inconsapevolmente, e per tutto il tempo che ancora passeremo insieme.

Ultima, ma non per importanza, vorrei dedicare un enorme ringraziamento a Giorgia. In questo anno e mezzo mi sei stata vicina, aiutandomi e tirandomi su di morale quando ero triste o mi lamentavo per ogni cosa. Le cenette fit in spiaggia, gli hamburger o la pizza mangiati davanti a qualche film o serie TV, le passeggiate nei parchi o anche solo sotto casa tua per far fare il giretto degli alberi a Dylan. Mi hai sempre fatto sentire sicuro di me stesso e in grado di fare tutto, se solo lo avessi voluto. Per tutto questo, ti sarò sempre grato.

Infine, aggiungo anche una parte auto-referenziale. Grazie a me stesso e al me di 10 anni fa, che sapeva benissimo cosa volesse dalla vita: studiare informatica per poter un giorno diventare un hacker. Beh, non ho mai hackerato nulla, ma almeno l'ambito di studio è rimasto quello. Guardando indietro, mi rendo conto di quanta strada ho percorso, tra inciampi e successi. Sono fiero di poter finalmente dire: 'Ce l'ho fatta'.

Finiti questi ringraziamenti, voglio aggiungere solo una cosa, un detto che rispecchia il mio percorso fino ad ora:

Abstract

The project discussed in the thesis pertains to a mixed-reality system designed to assist industrial operators in recognizing objects and providing guidance during assembly in the production chain. This task can be divided into two main steps: the development of a deep learning computer vision model and the development of the mixed-reality system. The deep learning model has been created taking inspiration from state-of-the-art research, while also considering the necessary constraints imposed by the final application context, such as high accuracy and robustness to occlusion. The mixed-reality step involves creating a junction between the deep learning model and the visor through the visor engine. This last step is fundamental to provide operators with a ready-to-go system.

Contents

1 Introduction **1**

2 State of the Art **3**
- 2.1 Object Detection . . . 3
  - 2.1.1 Object Detection using Deep Learning Models . . . 3
  - 2.1.2 Two-Stage Detectors and Milestones . . . 4
  - 2.1.3 One-Stage Detectors and Milestones . . . 6
- 2.2 Object Pose Estimation . . . 9
  - 2.2.1 Techniques and Challenges . . . 9
  - 2.2.2 Applications . . . 10
  - 2.2.3 Model-Based Algorithms and Milestones . . . 10
  - 2.2.4 Model-Free Algorithms and Milestones . . . 11
  - 2.2.5 History of Deep Learning-based 6DoF Pose Estimation Algorithms . . . 12
- 2.3 Datasets for Object Pose Estimation Tasks . . . 15
  - 2.3.1 Synthetic Data Generation . . . 17
- 2.4 Understanding Immersive Technologies . . . 18
  - 2.4.1 Augmented Reality . . . 18
  - 2.4.2 Virtual Reality . . . 19
  - 2.4.3 Mixed Reality . . . 19

3 First Approach: VideoPose **21**
- 3.1 YOLOv8 as Object Detection Module . . . 21
- 3.2 VideoPose Implementation . . . 22
- 3.3 Tests and Results . . . 23

4 System Architecture **29**
- 4.1 Use-Cases and Goals . . . 29
  - 4.1.1 Project Goal . . . 29
  - 4.1.2 Project Use-Cases . . . 30
- 4.2 PoET Architecture Overview . . . 30
- 4.3 Scaled-YOLOv4 Backbone . . . 30
  - 4.3.1 Knowledge Basis . . . 31
  - 4.3.2 Proposed Models . . . 32
- 4.4 PoET Transformer . . . 34
- 4.5 Visor engine and Communication . . . 35

5 System Implementation **38**
5.1 Project Structure Description 38
5.2 Synthetic Video Generation 38
5.2.1 Blender Scripting 38
5.2.2 Bounding Box Generation 43
5.2.3 YCB-Video BOP Format Conversion 45
5.2.4 YOLO Format Conversion 49
5.3 PoET Implementation 50
5.3.1 Scaled-YOLOv4 Implementation 50
5.3.2 PoET Transformer Implementation 54
5.4 Model Deployment 55
5.4.1 YOLO Inference Script 55
5.4.2 PoET Inference Script 56
5.4.3 AR Visor Script 56

6 Test and Results **60**
6.1 Experiments' Configuration 60
6.2 Experiments Analysis 63
6.2.1 PoET Metrics 63
6.2.2 YOLO Metrics 65
6.2.3 Experiment I and II (PoET Training) 67
6.2.4 Experiment III (PoET Training) 67
6.2.5 Experiment IV (PoET Training) 68
6.2.6 Experiment V (YOLO Training) 68
6.2.7 Experiment VI (PoET Training) 73
6.2.8 Experiment VII (YOLO Training) 74
6.2.9 Experiment VIII (PoET Training) 76
6.2.10 Experiment IX (YOLO Training) 78
6.2.11 Experiment X (PoET Training with YOLO) 80
6.2.12 Experiment XI (PoET Training) 81
6.2.13 Experiment XII (YOLO Training) 83
6.2.14 Experiment XIII (PoET Training with YOLO) 85
6.2.15 Experiment XIV (PoET Training with YOLO Fine-Tuning) . 86
6.2.16 Inference Experiment 88

7 Conclusions **90**

8 Future Work **91**

Bibliography **92**

Chapter 1

Introduction

In the fast-paced, technologically driven landscape of the industrial sector, the pursuit of efficiency and precision is critical to the success of proposed services. This thesis represents a collaborative effort between Sapienza University of Rome and Thales Alenia Space, a globally recognized leader in satellite systems and services. Thales Alenia Space, with its extensive expertise in the design, manufacturing, and delivery of satellite systems for applications such as telecommunications, navigation, Earth observation, and space exploration, provides invaluable industry insight and real-world challenges to this research endeavor.

The genesis of this project stemmed from the pressing need within industrial settings to enhance productivity. Currently, satellite operators are required to manually build and assemble parts, a process that is prone to errors. At each step, a quality inspector is introduced to identify and rectify potential mistakes. However, during satellite assembly, these interruptions for quality checks can significantly extend the time required to complete the entire operation. The primary goal of this project is to develop a tool that assists operators during satellite assembly, thereby reducing errors and improving productivity, while maintaining the same level of build quality. With the implementation of this system, the role of the quality inspector becomes redundant, as the system also supports the quality-check processes.

At the core of our project lies the primary objective of developing a deep learning framework capable of classifying objects and accurately estimating their position relative to the camera using only a single RGB image. While the field of 6D pose estimation has achieved remarkable advancements since 2017, with a plethora of research publications proposing novel methodologies and enhancements, the majority of state-of-the-art approaches rely on RGB-D data, leveraging depth information. Here lies one of the key challenges we were confronted with: implementing a methodology that exclusively utilizes RGB features for pose regression, thereby circumventing the reliance on depth data. In particular, one of the central tasks of this project, is the detection and localization of objects in three-dimensional (3D) space relying only on RGB images. While humans perform this task effortlessly through experience and spatial cognition, for machines, it presents a formidable challenge. Addressing this challenge requires the application of deep learning techniques to accurately infer both an object's class and position. These parameters are essential for the quality inspection process. If the system can successfully regress both the object's position and class, it will enable the development of a procedural algorithm based on the trained AI model. This algorithm would guide the operator through the assembly process while performing the quality inspection steps, ensuring that the final product adheres to both the building and flight constraints defined during the design phase.

Furthermore, our ambitions extend beyond the development of a robust 6D pose estimation model. The model's final predictions are not useful if not directly available to the operator. Therefore, the idea was to equip the operators with a mixed-reality visor, such that the needed information can be plotted into the visor display. This futuristic approach enables operators to access information in an easier and faster way, thus revolutionizing their interaction with industrial environments. However, the computational demands imposed by deep learning models present a substantial challenge for standalone deployment. As a result, our model carefully balances the need for high accuracy with the requirement for optimized performance, particularly in terms of inference time (a critical consideration for real-world applications).

Chapter 2

State of the Art

2.1 Object Detection

When we talk about "object detection" in computer vision, we are typically referring to tasks where the goal is to identify whether an object from a given set of categories is present in an image (or video) frame. For a computer, this task can be much more challenging than it appears to a human. We are able to recognize objects intuitively, without needing to consciously think about it. This process involves several automatic steps, including image acquisition through the eyes, image processing using various brain areas, and, ultimately, applying our experience to understand what we are seeing. For an algorithm, all these steps represent numerous implicit challenges, such as understanding how to recognize an object, creating the bounding boxes[1], or even deducing context to make more reliable predictions.

2.1.1 Object Detection using Deep Learning Models

Deep learning, a subset of machine learning, is a branch of artificial intelligence that focuses on the development and training of neural networks with multiple layers to learn representations of data. These neural networks, inspired by the structure and function of the human brain, are capable of automatically learning hierarchical patterns and features from raw input data, such as images, audio, text, and sequences.

Deep learning models, often referred to as deep neural networks, consist of interconnected layers of neurons, each performing specific transformations on the input data. The depth of these networks, referring to the number of layers, enables them to learn complex and abstract representations of data. This leads to state-of-the-art performance in various tasks such as image recognition, natural language processing, speech recognition, and more.

The learning process in deep learning involves training the neural network on a large dataset, adjusting the parameters of the network through optimization algorithms such as gradient descent, and iteratively refining the model's predictions to minimize the difference between predicted and actual outputs.

Deep learning has completely revolutionized the object detection task, achieving performance levels never previously reached and setting a new starting point for the state-of-the-art.

[1] In computer vision, a bounding box is the area in the image that contains the object. Typically, these boxes are represented using four corners of the box, or two opposite corners, with the other two corners implicitly deduced.

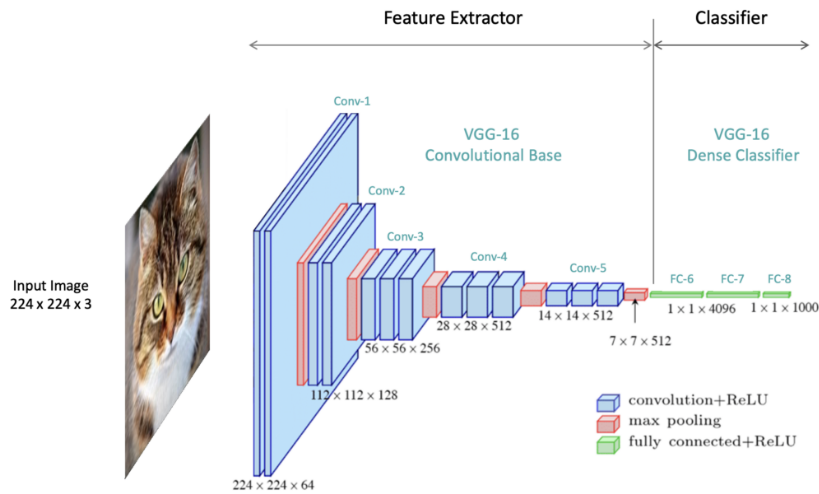


Figure 2.1. CNNs consist of convolutional layers, which apply learnable filters to input data, detecting features like edges or textures. Pooling layers then down-sample the feature maps produced by convolution, retaining essential information. Fully connected layers follow, connecting neurons across layers for classification or regression.

In general, we can organize each object detection deep learning model into two main categories: two-stage detectors (Section 2.1.2) and one-stage detectors (Section 2.1.3) [1] [2]. Convolutional Neural Networks [3] (CNNs Figure 2.1) are considered the core of both detector categories. CNNs automatically learn features from input data using convolutional layers, which apply filters to capture patterns. Pooling layers down-sample feature maps, and fully connected layers integrate features for classification or regression tasks. CNNs excel in recognizing patterns in images, making them ideal for object detection.

2.1.2 Two-Stage Detectors and Milestones

The two-stage architecture [4], known for its higher accuracy, is relatively more complex compared to one-stage algorithms. In the first stage, often referred to as the proposal stage, preliminary tests are performed on the input image using a region proposal network (RPN). The RPN generates candidate bounding boxes, also known as regions of interest (RoIs), where objects might be present. These candidate RoIs are identified based on their likelihood of containing objects of interest.

In the second stage, commonly known as the refinement stage, the algorithm performs regional classification and location refinement on the RoIs generated in the previous stage. Each RoI is processed independently to classify the object category it contains and refine the bounding box coordinates for accurate localization. This refinement typically involves fine-tuning the coordinates of the bounding box to better align with the object's precise location within the RoI.

Some commonly used architectures in the two-stage detection framework include Faster R-CNN, R-FCN, and FPN. These architectures are usually applied in scenarios where high accuracy is crucial, such as in medical imaging or autonomous driving applications.

Fast R-CNN Proposed in 2015 by Girshick *et al.* [5], Fast R-CNN is an advancement in object detection, built on the R-CNN framework. Unlike earlier models that

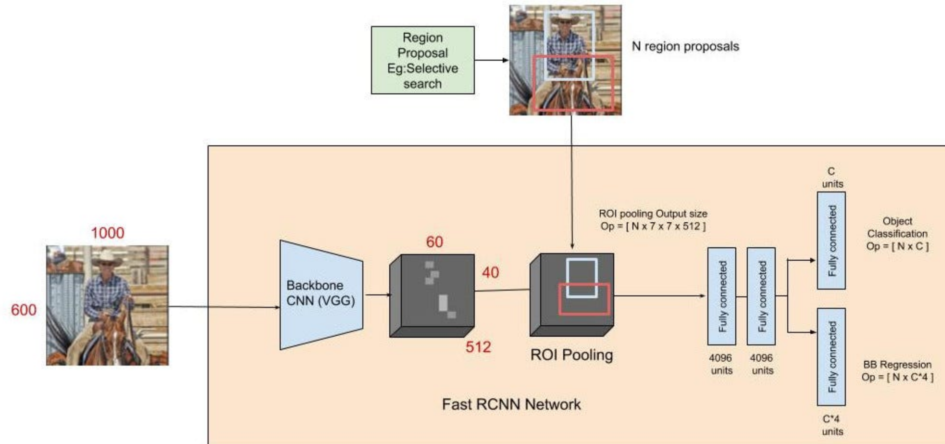


Figure 2.2. Faster R-CNN Architecture.

relied on external algorithms, like selective search for region proposal generation, Fast R-CNN was the first to incorporate an internal Region Proposal Network (RPN). This network efficiently proposes regions likely to contain objects, eliminating the need for computationally expensive external methods. Once the region proposals are obtained, Fast R-CNN utilizes a single-stage CNN to extract feature maps from the entire image, capturing rich information about the image as a whole. To ensure each region proposal is mapped to a fixed-length feature vector, regardless of its size or aspect ratio, Fast R-CNN introduces the Region of Interest (RoI) pooling layer. This layer extracts fixed-size feature maps from the convolutional feature maps for each region proposal. After RoI pooling, the fixed-size feature vectors are passed through fully connected layers for both classification (predicting object classes) and bounding box regression (refining object localization). These layers together make the final predictions regarding the presence of objects within each region and their respective bounding box coordinates.

Faster R-CNN Proposed by Ren *et al.* the same year as Fast R-CNN, Faster R-CNN [6] builds on the innovations introduced by its predecessor. It further improves object detection efficiency by integrating the Region Proposal Network (RPN) directly into the detection framework. This unified architecture allows for end-to-end training and inference, enabling Faster R-CNN to generate region proposals and perform object detection in a single forward pass. By sharing convolutional features between the RPN and the detection network, Faster R-CNN achieves significant computational savings, making it faster than Fast R-CNN. Additionally, Faster R-CNN introduces anchor boxes—predefined boxes of various scales and aspect ratios used by the RPN to efficiently propose regions. This technique improves localization accuracy and enhances overall object detection performance (Figure 2.2).

Feature Pyramid Networks (FPN) Published in 2017 by Lin [7], Feature Pyramid Networks (FPN) address the challenge of detecting objects at multiple scales within an image. FPN constructs a top-down architecture with lateral connections, enabling it to generate a pyramid of feature maps with semantic information at

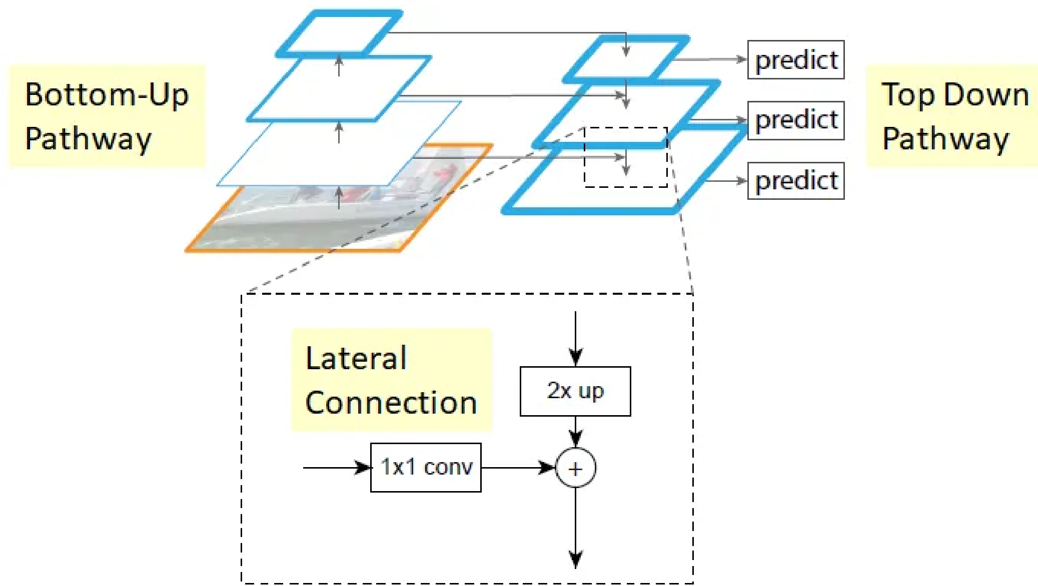


Figure 2.3. FPN Architecture.

different resolutions. This design allows FPN to capture and utilize information at various scales, facilitating robust object detection across different object sizes. By incorporating FPN into detection frameworks, such as Faster R-CNN, detectors can effectively detect both small and large objects, leading to improved detection performance (Figure 2.3).

Mask R-CNN Introduced in 2017 by He *et al.*[8], Mask R-CNN is an extension of Faster R-CNN that adds a branch for predicting segmentation masks on each Region of Interest (RoI), in parallel with the existing branch for bounding box recognition. This architecture enables the model to perform both object detection and instance segmentation simultaneously, which is useful for tasks that require precise localization of objects within an image. The additional mask output is distinct from bounding box detection because it provides a per-pixel segmentation of the object, which is more detailed than just the rectangular box. In the first stage, Mask R-CNN scans the image and generates proposals about the regions where there might be an object based on the feature maps created by a backbone network like ResNet [9]. The second stage, the RoI Align, extracts these proposals and performs precise object classification, bounding box regression, and mask prediction. This method allows for highly accurate object detection and segmentation, even in challenging scenarios where objects are closely clustered or when precise object boundaries are required.

2.1.3 One-Stage Detectors and Milestones

One-stage detectors [10], characterized by their simplicity and efficiency, perform object detection in a single step without explicit proposal generation. Instead of separating the detection into multiple stages, one-stage detectors directly predict bounding boxes and class probabilities for objects across the entire image.

In these algorithms, the network typically divides the input image into a grid of cells and predicts bounding boxes and class probabilities for objects within each

cell. This approach enables one-stage detectors to achieve real-time performance and simplicity compared to their two-stage counterparts.

Examples of widely used one-stage detectors include YOLO, SSD, RetinaNet and DETR. These models are known for their speed and efficiency, making them suitable for applications requiring fast inference, such as video surveillance, autonomous drones, and real-time object tracking.

YOLO-Family YOLO (You Only Look Once) [11] is a pioneering series of object detection models known for their simplicity and real-time performance. YOLO approaches object detection as a single regression problem, directly predicting bounding boxes and class probabilities for objects in images.

YOLO v1 Introduced by Redmon *et al.* in 2016 [12], YOLO v1 divides the input image into a grid and predicts bounding boxes and class probabilities for each grid cell. It utilizes a single convolutional network to make predictions across the entire image, achieving real-time performance (Figure 2.4a).

YOLO v2 Released in 2017, YOLO v2 improves upon its predecessor by incorporating various enhancements, including batch normalization, high-resolution classifiers, anchor boxes for improved localization, and multiscale training. These improvements lead to better accuracy and stability.

YOLO v3 YOLO v3, unveiled in 2018, introduces further improvements such as feature pyramid networks (FPN), enabling the model to detect objects at different scales. It also adopts a variant of Darknet, the network architecture used in YOLO, with additional convolutional layers and skip connections for better feature extraction.

YOLO v4 YOLO v4, released in 2020, brings significant advancements in accuracy and speed. It introduces novel techniques such as CSPDarknet53 as the backbone, path aggregation networks (PAN), and spatial pyramid pooling (SPP) for better feature representation and context aggregation. YOLO v4 achieves state-of-the-art performance while maintaining real-time inference speeds.

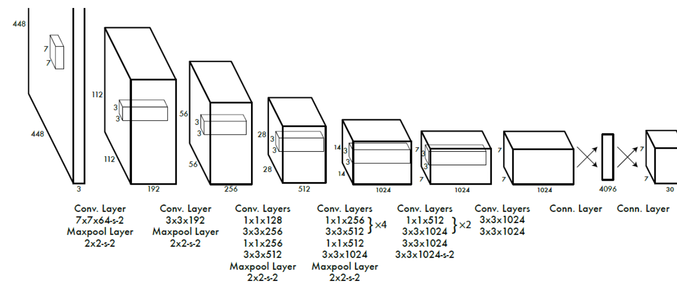
YOLO v5 YOLO v5, introduced in 2020, is developed by Ultralytics. It is not an official release by the original YOLO authors but has gained popularity for its simplicity and effectiveness. YOLO v5 adopts a lightweight architecture based on the CSPNet and focuses on simplicity, speed, and ease of use. Despite its smaller model size, YOLO v5 achieves competitive accuracy compared to previous versions.

YOLO v6 Introduced in 2022, YOLOv6 represents a refinement of the YOLO trunk and neck. This version introduces the EfficientRep Backbone and Rep-PAN Neck, tailored for enhanced efficiency. Unlike previous iterations like YOLOv5, where the classification and box-regression heads leverage the same features, YOLOv6 adopts a distinct approach. Here, additional layers are introduced to segregate these features from the final head, resulting in improved performance.

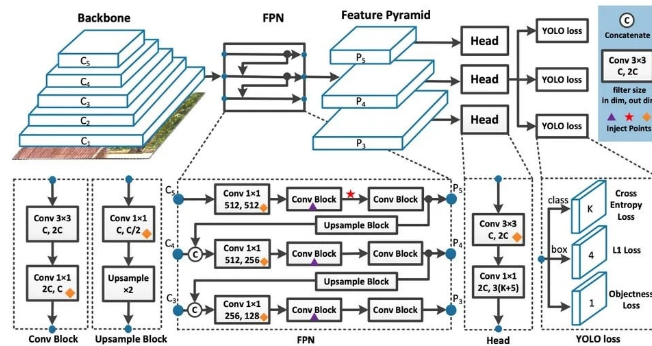
YOLO v7 Released in 2022, YOLOv7 does not really implement new features or strategies.

YOLO v8 YOLO v8 was published in early 2023, it proposes a new backbone network, anchor-free detection head, and loss function. This has lead to a performance boost not only for the higher-level GPUs but also for the lower-end devices such as CPU and older GPUs (Figure 2.4b).

Single Shot Multi-Box Detector (SSD) SSD [13] was proposed by W. Liu *et al.* in 2015. The main contribution of SSD is the introduction of multi-reference (pre-defined bounding boxes) and multi-resolution (down-sampling of the feature maps over the different layers, a typical approach adopted in CNNs) detection techniques, which significantly improve the detection accuracy of a one-stage detector, especially



(a) YOLOv1 Architecture.



(b) YOLOv8 Architecture.

Figure 2.4. As we can see from the two schemes above, the YOLO architectures changed a lot during these years.

for small objects. A key difference between SSD and previous detectors is that SSD detects objects of different scales using reference boxes on different layers of the network, whereas previous detectors typically only run detection on their top layers

RetinaNet Proposed in 2017 by Lin *et al.*, RetinaNet addresses the challenge of class imbalance in object detection by introducing a novel focal loss function. This loss function dynamically adjusts the weight assigned to each training example based on its classification difficulty, focusing more on hard examples while reducing the influence of easy ones. RetinaNet utilizes a feature pyramid network (FPN) backbone to extract multiscale features and employs a single-stage detection framework. By combining feature maps from different pyramid levels, RetinaNet achieves robustness to objects of varying sizes.

DETR Presented in 2020 by Carion *et al.* [14], DETR (DEtection TRansformers) represents a significant departure from traditional object detection methods. Instead of relying on anchor boxes or region proposal networks, DETR employs a transformer architecture (Figure 2.11), originally developed for natural language processing tasks, for end-to-end object detection. This approach treats object detection as a set prediction problem, where the model simultaneously predicts the set of object bounding boxes and their corresponding class labels. By directly optimizing a bipartite matching loss between predicted and ground truth boxes, DETR eliminates the need for heuristic-based methods for box prediction. This results in a simple yet effective approach that achieves competitive performance on standard object detection benchmarks (Figure 2.5).

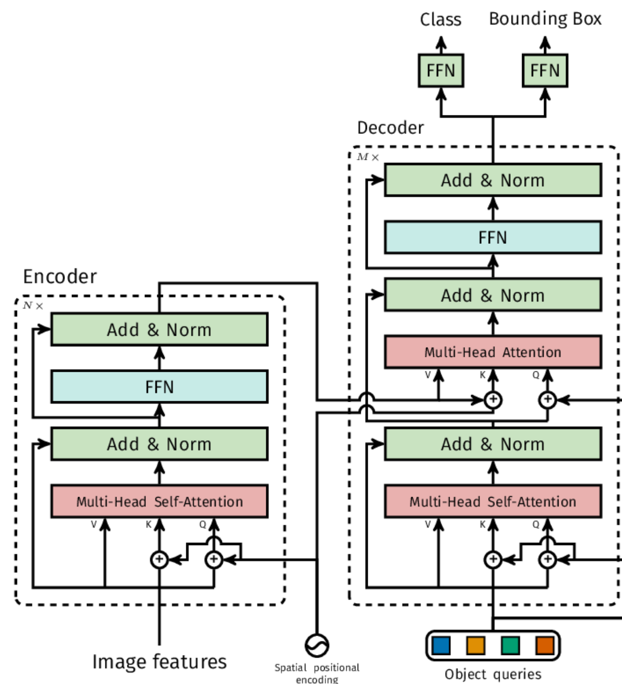


Figure 2.5. DETR Architecture.

2.2 Object Pose Estimation

Object pose estimation [15] can be considered the next-level task after object detection. While classical object detection focuses on locating objects within a frame, estimating the pose of an object involves representing its translation and rotation in real-world coordinates with respect to a fixed origin, typically the camera. To perform accurate pose estimation, the process relies on precise object detection. With knowledge of the object and its Region of Interest (RoI), pose prediction models can concentrate their efforts on accurately predicting the six degrees of freedom (6DoF)[2] of the object. This field is currently one of the most studied in computer vision, especially for industrial fields.

2.2.1 Techniques and Challenges

Object pose estimation techniques are generally classified into two major categories: model-based (Section 2.2.3) and model-free (Section 2.2.4) approaches. Model-based methods require a prior knowledge of the object's 3D model. They often utilize matching techniques that compare observed features from the input image against a database containing the 3D model's features, employing algorithms like Iterative Closest Point (ICP) for refinement. On the other hand, model-free approaches infer pose directly from image data using machine learning models, particularly deep learning techniques which do not necessarily require a predefined model of the object.

[2]The "6DoF" of an object, often referred to as the six degrees of freedom, describe its potential movements or transformations in three-dimensional space. These degrees encompass translation along the x, y, and z axes, representing horizontal, vertical, and depth movements respectively. Additionally, rotation around each of these axes allows for pitch, roll, and yaw motions. This concept is fundamental in fields such as robotics, computer vision, and virtual reality.

2.2.2 Applications

Pose estimation is fundamental in numerous applications, including robotic manipulation, autonomous driving [16], augmented reality and gaming [17], and human-computer interaction. For instance, in robotic manipulation, accurate pose estimation allows robots to interact with objects in their environment more effectively. In the domain of autonomous vehicles, the estimation of the pose of nearby vehicles and pedestrians is crucial for safe navigation and interaction.

2.2.3 Model-Based Algorithms and Milestones

Model-based pose estimation methods rely heavily on pre-existing knowledge of the 3D models of the objects being identified. These approaches match the observed data from the sensors or images with the 3D model stored in a database.

Iterative Closest Point (ICP) The Iterative Closest Point (ICP) algorithm revises the estimated pose iteratively to minimize the distance between the 3D points of the model and the data collected from the real world. It is recognized for its simplicity and robustness, achieving high accuracy when the initial approximation of the object's pose is known. However, its performance declines in the presence of noise and outliers, and its computational demands can preclude real-time applications. A notable milestone for ICP was its application in the Mars Rover missions, aiding in 3D mapping and navigation.

Feature-Based Matching This method utilizes distinctive features from the object's 3D model, such as edges and corners, matching these with observed features in the image. It is particularly robust to partial occlusions and varying lighting conditions, making it preferable over direct image matching. These algorithms depend on the quality and selection of features; poor-quality features can lead to inaccuracy. Enhancements in feature extraction techniques, like SIFT[3] [18] and ORB[4] [19], have bolstered its use in complex scenes.

Template-Based Methods In template-based methods, observed object views are compared against a set of pre-rendered templates of the 3D model from various angles. These methods excel in environments where the object's appearance variations are minimal and well-modeled beforehand. However, they lack flexibility to handle new objects not in the template set and require extensive computational resources due to numerous comparisons. They have seen practical applications in industrial quality control, where objects adhere to strict standards.

PnP (Perspective-n-Point) PnP algorithms determine the pose of an object by establishing spatial positions relative to a set of 3D points and their corresponding 2D projections in the image. This approach is efficient with a limited number of

[3]SIFT (Scale-Invariant Feature Transform): SIFT is a feature detection algorithm used in computer vision to detect and describe local features in images. It identifies keypoints that are invariant to scale, rotation, and illumination changes, making it robust for various applications such as object recognition and image stitching.

[4]ORB (Oriented FAST and Rotated BRIEF): ORB is a feature detection and description algorithm designed for real-time applications. It combines the FAST keypoint detector with the BRIEF descriptor and introduces rotation invariance by computing orientations for keypoints. ORB is efficient, robust to noise, and suitable for tasks like object tracking and localization on resource-constrained devices.

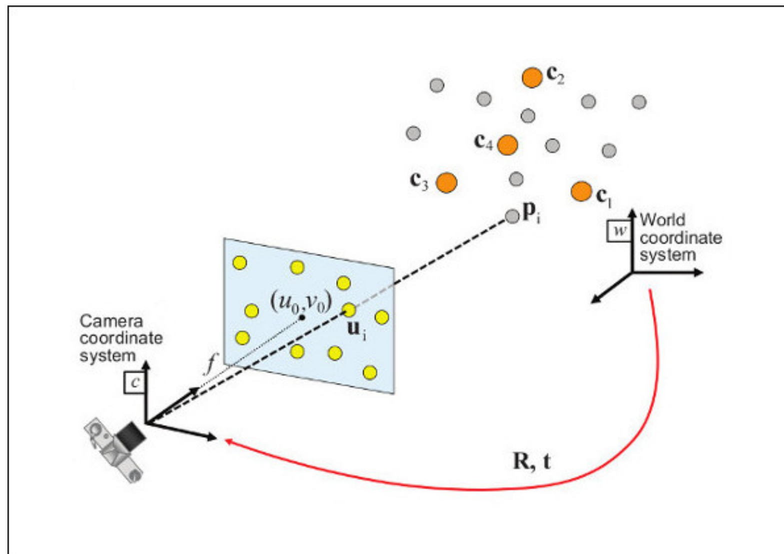


Figure 2.6. PnP point alignment logic.

correspondences, but its accuracy is contingent upon correct matches between 3D and 2D points, which is challenging in cluttered scenes. Recent advances in solving PnP problems have propelled significant enhancements in real-time augmented reality applications, where quick pose estimation is crucial (Figure 2.6).

2.2.4 Model-Free Algorithms and Milestones

Model-free pose estimation approaches do not rely on prior knowledge of the 3D models of objects. Instead, these methods infer pose directly from image data, typically using advanced machine learning models. This category of approaches is particularly adaptable to diverse and novel object types.

Deep Learning Based Methods Leveraging the power of deep neural networks, especially convolutional neural networks (CNNs), this approach has revolutionized the field of pose estimation. Deep learning methods automatically extract and learn the most relevant features for pose estimation from vast amounts of data. While highly effective in handling complex and varied data inputs, these methods require substantial computational resources and large labeled datasets for training, which can be a limiting factor.

Regression Forests Regression forests approach pose estimation by using an ensemble of decision trees to regress object poses directly from pixel values. This method is efficient and capable of real-time performance on standard hardware, making it suitable for applications like motion capture in gaming. However, its performance can degrade with increased object variability unless specifically trained for generalization. A significant advancement was its application in Microsoft Kinect, where it enabled real-time human body pose estimation with high accuracy.

Template Matching with Learned Features Unlike traditional template-based methods that rely on predefined templates, this approach uses machine learning to dynamically learn templates from the training data. This flexibility allows it to

effectively deal with a wide range of objects and poses. While this method benefits from not requiring a rigid set of templates, it still faces challenges in environments with significant background clutter or when objects appear at drastically different scales. It has been effectively used in robotic systems to recognize and manipulate objects in unstructured environments.

Unsupervised and Semi-supervised Learning Techniques These techniques are particularly appealing in scenarios where labeled data is scarce or expensive to obtain. By using unlabeled data, these methods can reduce the reliance on extensive labeled datasets, although they typically achieve lower accuracy compared to supervised methods. Recent developments include the use of generative adversarial networks (GANs) to augment pose estimation datasets, enhancing the robustness of pose estimation models under varied conditions.

2.2.5 History of Deep Learning-based 6DoF Pose Estimation Algorithms

Since 2012 a lot of research papers about object pose estimation have been published, not only for industrial purposes but also for surveillance, video games, autonomous driving, etc... One of the first and more relevant papers about deep learning approach is *"Model Based Training, Detection and Pose Estimation of Texture-Less 3D Objects in Heavily Cluttered Scenes"* published by Hinterstoißer *et al.* in 2012 [20]. Their key idea was to learn a projection function able to map the data points into one shared and several private latent spaces with an orthogonality constraint between them. Another important milestone is represented by *"Learning 6D Object Pose Estimation Using 3D Object Coordinates"* published by Brachmann *et al.* in 2014[21]. In this paper, the approach consisted in using a single decision forest [5] to classify each pixel from an RGB-D image. Once that the object was obtained using a voting system, an energy function optimization problem was used to estimate the 6DoF of the object.

PoseCNN In 2017, the NVIDIA Research group in collaboration with Washington University published *PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes* [22]. PoseCNN employs a unique approach by breaking down the pose estimation task into separate components, such as rotation and translation. This design allows the network to effectively manage the relationships between these components (Figure 2.7). Initially, the network predicts an object label for each pixel in the input image. Subsequently, it calculates the 2D pixel coordinates of the object center by predicting unit vectors from each pixel to the center. Leveraging semantic labels, pixels associated with an object contribute to determining the object's center location through voting mechanisms. Furthermore, the network estimates the distance from the object center. With known camera intrinsic[6], this information facilitates the recovery of the object's 3D translation. Finally, the network estimates the 3D rotation by regressing convolutional features extracted within the object's bounding box to a quaternion representation. Notably,

[5] A single decision forest is a fundamental machine learning model used for classification and regression tasks. Decision trees recursively partition the input space into regions based on feature values, assigning labels or predicting values for each region.

[6] In photography and computer vision, camera intrinsic parameters refer to internal characteristics of a camera that are independent of external factors such as scene geometry or lighting conditions. These parameters include focal length, principal point coordinates, and lens distortion coefficients, which collectively define how light rays are projected onto the camera's image sensor.

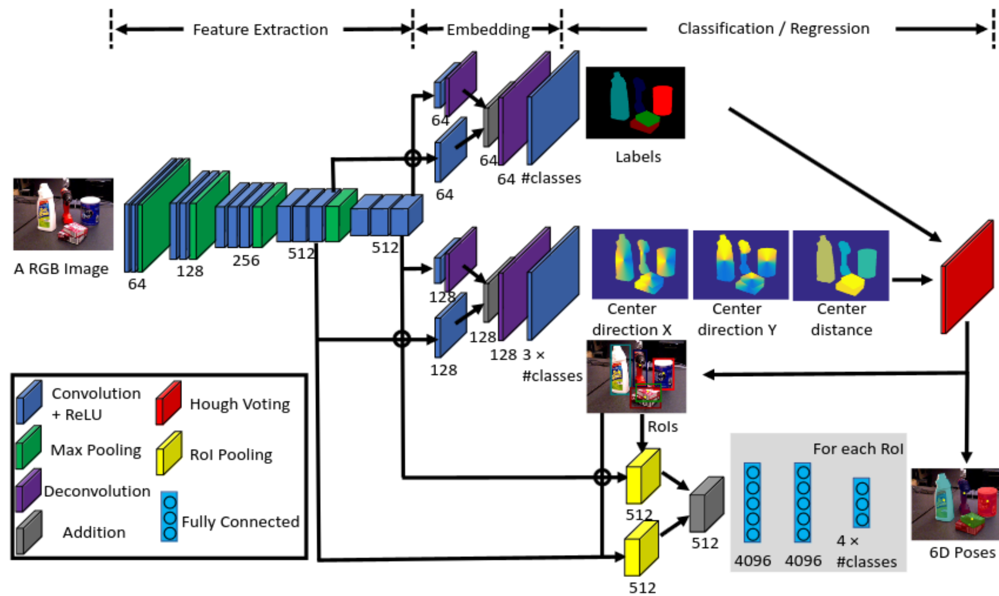


Figure 2.7. PoseCNN architecture.

this approach of 2D center voting followed by rotation regression proves effective for both textured and texture-less objects and remains robust against occlusions, as the network is trained to infer object centers even when they are partially obscured. This model was so innovative that it is still used for comparison in state-of-the-art publications.

DeepIM The subsequent year, 2018, Yu Xiang *et al.* proposed a novel approach based on the Deep Iterative Matching [23]. The key idea behind DeepIM is to formulate the pose estimation problem as an optimization task. The network iteratively refines the initial pose estimate by iteratively matching image features with object model features. This iterative refinement process enables the network to gradually improve the accuracy of the pose estimation. The network architecture consists of two main components: a pose refinement network and a matching network. The pose refinement network takes an initial pose estimate and an input image as input and generates a refined pose estimate. The matching network matches features extracted from the input image with features from a 3D object model to compute a similarity score, which is used to guide the pose refinement process (Figure 2.8).

PVN3D Wen Gu *et al.* in 2019 proposed PVN3D [24] for estimating the 6DoF pose of objects from RGB-D images. The main innovation was about the point-wise 3D keypoints voting mechanism to regress the object pose. PVN3D takes an RGB-D image as input and extracts features using a CNN. It leverages both RGB and depth information to capture rich visual features. After that feature generation step, PVN3D generates a set of 3D keypoints for the object in the scene and assigns each keypoint a set of votes indicating its potential position in 3D space. These votes are generated based on the extracted features and are used to estimate the object's pose. Finally, the system uses a regression network to predict the 6DoF pose of the object by aggregating the votes from the keypoints. The network learns to estimate the object's translation and rotation parameters directly from the voting results

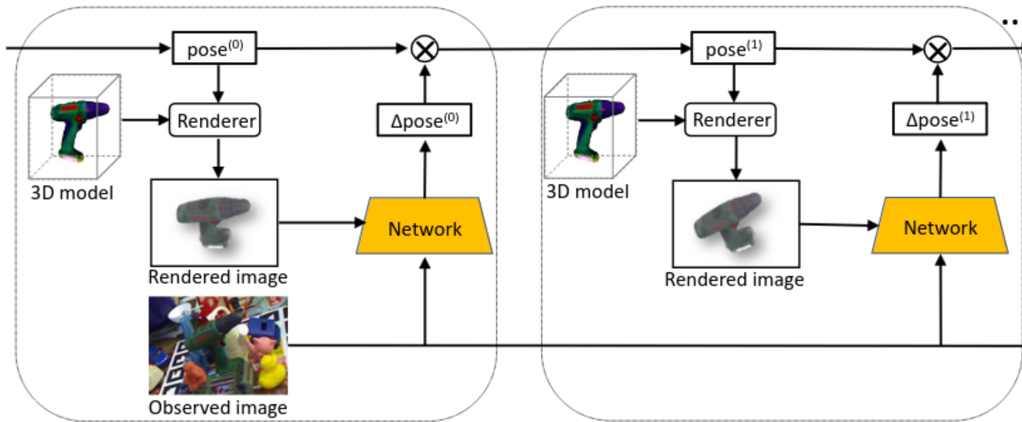


Figure 2.8. DeepIM network.

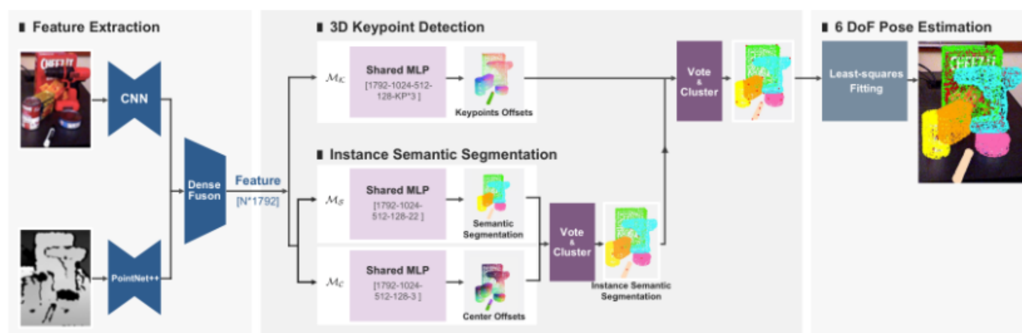


Figure 2.9. PVN3D network.

(Figure 2.9).

MaskedFusion One of the first approaches published in 2020 was MaskedFusion [25] by Pereira *et al.*, a sophisticated architecture for 6D object pose estimation from RGB-D images. It begins by generating high-quality instance-level object masks, leveraging both RGB and depth information. These masks precisely delineate object boundaries, crucial for accurate localization. The method then fuses RGB features with mask features to enhance discriminative power. This fusion process ensures that the network focuses on relevant object regions while suppressing background clutter. The fused features are passed through a neural network for pose regression. Trained end-to-end, this network optimizes pose estimation performance. Additionally, MaskedFusion employs a novel loss function tailored for mask-based pose estimation. It effectively handles occlusions and cluttered backgrounds, further improving accuracy (Figure 2.10).

Vision Transformer (ViT) In 2020 Dosovitskiy *et al.* came up with the paper *"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale"* [26], this approach applied the Transformer architecture (Figure 2.11), presented in *"Attention is all you need"* by Vaswani *et al.* in 2017 [27], to a computer vision task. The ViT architecture (Figure 2.12) represented a significant departure from conventional convolutional neural networks (CNNs). By adopting the transformer

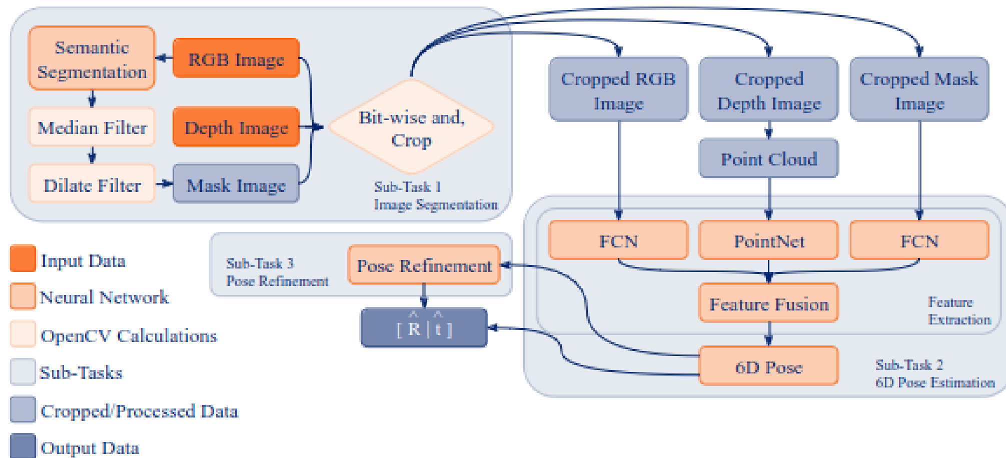


Figure 2.10. MaskedFusion network.

architecture originally designed for natural language processing tasks, ViT redefined the general way on how we approach image classification challenges. Rather than relying on traditional CNN methodologies of convolutions and pooling operations, ViT revolutionizes image processing by dividing images into fixed-size patches. These patches are treated as sequences of tokens, like the words in natural language, and are subsequently fed into a transformer model. Through this novel approach, ViT facilitates parallelized processing and excels at capturing long-range dependencies within images. Notably, ViT's transformation of spatial hierarchies into self-attention mechanisms has resulted in impressive performance on various image classification benchmarks. Remarkably, ViT achieves competitive results with significantly fewer parameters compared to conventional CNN-based models, demonstrating its potential for efficient and effective image recognition. Nowadays, most of the novel approaches presented in computer vision conferences utilize ViT or its derivatives to address challenges and tasks such as pose estimation and object detection.

2.3 Datasets for Object Pose Estimation Tasks

Very complicated models require a huge amount of data to be trained and to achieve good performance. A dataset, in order to be used for object pose estimation tasks, needs to be labeled in such a way that the model can attempt to make predictions and adjust the parameters using the ground truth labels. Therefore, the labels used in this field should generally comprise 6DoF ground truth poses of the objects for each frame. The main datasets used for this purpose are:

- LineMod dataset (LM) and its next version LineMod Occluded (LMO) [28] (2015)
- T-LESS dataset [29] (2017)
- BOP-Datasets (a collection of other well-known datasets) [30] (2018)
- HomebrewedDB [31] (2019)
- HOPE by NVIDIA [32] (2019)

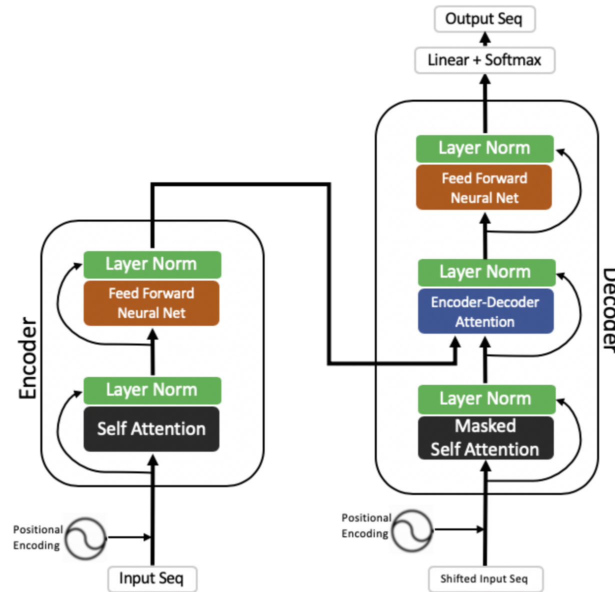


Figure 2.11. Transformer architecture. Transformers revolutionized various tasks, including language translation, text generation, and image captioning, by enabling parallelization, capturing long-range dependencies, and facilitating attention mechanisms. Unlike recurrent neural networks (RNNs) or convolutional neural networks (CNNs), transformers rely solely on self-attention mechanisms, allowing them to process entire sequences in parallel, making them highly efficient for long sequences. Transformers consist of an encoder-decoder architecture, where the encoder processes input sequences, and the decoder generates output sequences.

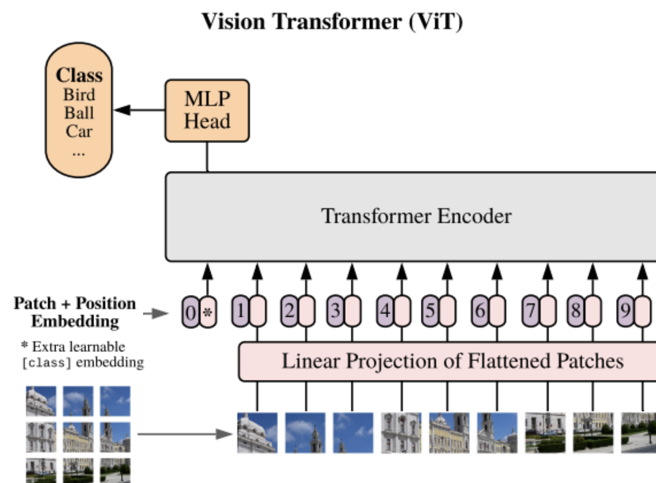


Figure 2.12. ViT architecture.

- YCB-Video dataset [33] (2020)

These are only some of the most important datasets used in the object pose estimation task.

2.3.1 Synthetic Data Generation

Some above-listed datasets (such as YCB-Video and T-LESS) not only provide images with annotations but also include 3D CAD models[7] of the objects on which the pose estimation should be performed. State-of-the-art models tend to utilize these provided CAD models as an additional resource during training to enhance the deep learning model's performance. Moreover, if the object models on which the prediction should be performed are not included in the dataset, we cannot use the trained model on new objects. Additionally, creating a dataset for estimating objects' 6DoF can be very challenging and could require years to complete. Consequently, for industrial applications, this approach is often discarded in favor of synthetic data generation. In the realm of data science and machine learning, synthetic data refers to artificially generated data that mimics real-world data patterns and characteristics. Unlike real data collected from observations or measurements, synthetic data is created using algorithms or simulation techniques. These methods aim to replicate the statistical properties and underlying structures of the original data without disclosing sensitive or proprietary information.

Most of the time, using synthetic data is a way to overcome the lack of data for training, achieving more flexibility and scalability. However, they also require a more meticulous selection to avoid non-reliable data, which, if fed into a deep learning model, could produce non-realistic results. The general goal of deep learning models is typically to produce predictions over real non-labeled data, therefore synthetic data must be produced in a way that they can accommodate this task.

There are many techniques used to generate synthetic data for object-pose-estimation starting from a 3D CAD model. In 2017, Planche *et al.* proposed the research paper *"DepthSynth: Real-Time Realistic Synthetic Data Generation from CAD Models for 2.5D Recognition"* [34]. The goal of the article was to generate synthetic depth images to emulate the depth produced by real sensors. This approach has been used in subsequent papers such as [35], [36], and [37].

In 2019, Tremblay *et al.* published *"Falling Things: A Synthetic Dataset for 3D Object Detection and Pose Estimation"* [38] (Figure 2.13). Their approach involved creating a custom Unreal Engine 4 (UE4)[8] plugin capable of capturing screenshots in three different environments: a kitchen, a sun temple, and a forest. In each environment, objects were placed in well-defined positions, while camera orientation was randomly generated by selecting from predefined poses.

Another very popular approach is to use Blender[9] for image rendering. This

[7]Computer-aided design (CAD), a CAD model refers to a digital representation of a physical object or system created using specialized software. CAD models encompass detailed geometric information about the object's shape, dimensions, and structural features.

[8]Unreal Engine is a widely-used game engine developed by Epic Games. Originally introduced in 1998, the Unreal Engine has since evolved into a comprehensive suite of tools and technologies for creating high-fidelity, real-time experiences across various platforms, including video games, virtual reality (VR), augmented reality (AR), architectural visualization, and film production. The Unreal Engine provides a robust framework for rendering three-dimensional environments, handling physics simulations, managing audio effects, and implementing complex gameplay mechanics.

[9]Blender is an open-source software suite renowned for its comprehensive set of tools and capabilities. Developed by the Blender Foundation, Blender offers a wide range of features for creating, editing, and rendering three-dimensional content across various domains, including animation, visual effects, game development, and architectural visualization.

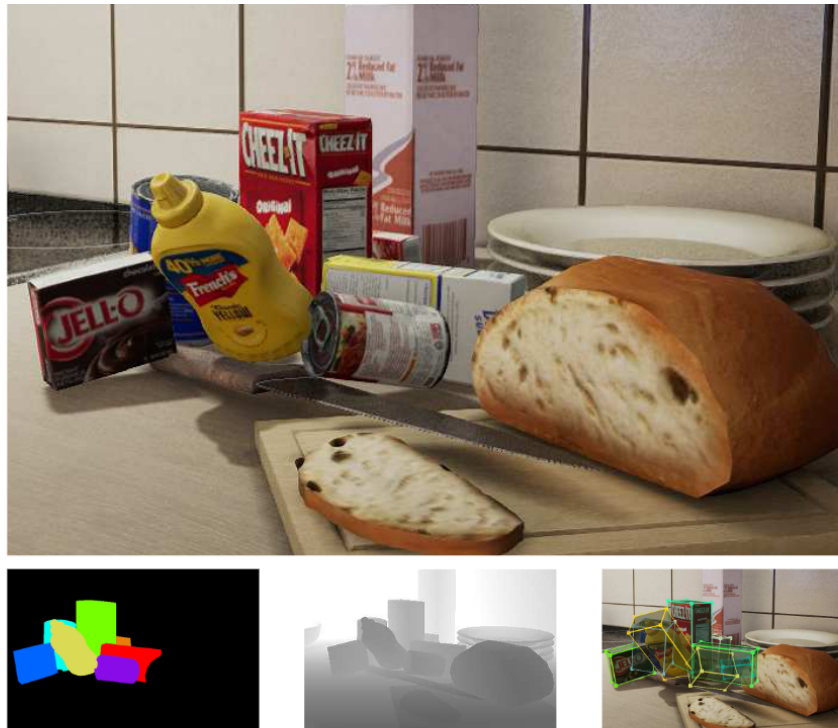


Figure 2.13. Example of synthetic data generation. Image taken from *"Falling Things: A Synthetic Dataset for 3D Object Detection and Pose Estimation"* [38]

.

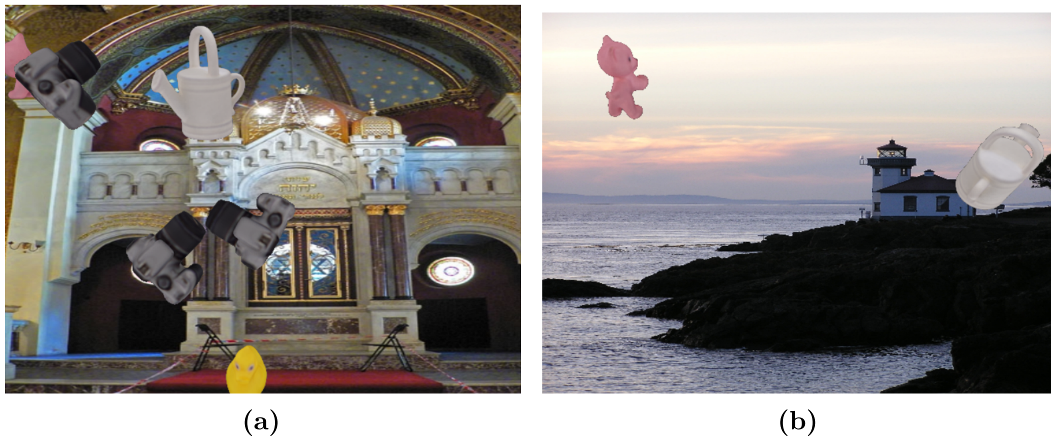
methodology typically involves generating random scenes in which the objects of interest are randomly placed, along with additional objects to simulate occlusion and random noise. [39][40][41][42] (Figure 2.14)

2.4 Understanding Immersive Technologies

In this project, as announced in the introduction, we leverage Mixed Reality technology for creating a tool that can be used in the industrial chain. To provide a definition of Mixed Reality (MR), we first have to define what Virtual Reality (VR) and Augmented Reality (AR) are.

2.4.1 Augmented Reality

Augmented Reality (AR) is a technology that integrates digital elements, such as images, videos, or 3D models, into the real-world environment. It enhances the user's perception of reality by overlaying virtual elements onto the physical world in real-time. This augmentation is interactive, offering users additional information, context, or experiences. AR can be experienced through various devices, including AR visors (e.g., Meta Quests, HTC Vive, Microsoft HoloLens) as well as smartphones and AR glasses. Popular examples of AR applications include Snapchat filters and Pokémon GO.



(a) (b)

Figure 2.14. Examples of synthetic data generation. Images taken from *"Generating Synthetic Data for Evaluation and Improvement of Deep 6D Pose Estimation"*[40]

2.4.2 Virtual Reality

Virtual Reality (VR) creates entirely immersive, computer-generated environments that replace the real world. Users typically wear VR headsets to block out their physical surroundings and immerse themselves entirely in the virtual environment. VR experiences can range from gaming and simulations to virtual tours and training exercises.

2.4.3 Mixed Reality

Mixed Reality (MR) combines elements of both AR and VR, allowing virtual objects to interact with the real world and vice versa. It integrates virtual elements into the user's physical environment, enabling interaction with virtual objects while maintaining awareness of the real world. MR experiences often involve spatial mapping and tracking technologies to blend virtual and physical elements seamlessly. Like VR, MR requires a visor to be used effectively.

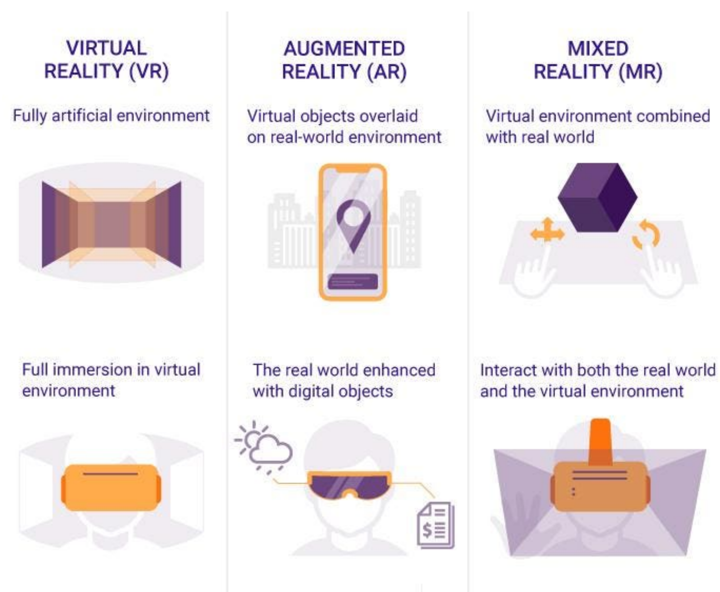


Figure 2.15. Image published on Forbes to explain the main differences between VR, AR and MR.

Chapter 3

First Approach: VideoPose

The first approach we tried to start with was VideoPose, introduced by Beedu *et al.* in 2021, the VideoPose framework [43] offers a novel approach to 6D object pose estimation utilizing a transformer architecture optimized for video sequences. The framework specifically leverages the sequential nature of video to improve prediction accuracy over time through a structured network that processes and predicts pose transformations frame by frame.

The primary components of VideoPose (Figure 3.1) include a Transformer for encoding visual features from sequential RGB-D frame patches, a Pose Regressor, and a Future Feature Predictor. The Transformer module, based on the Swin Transformer [44] or BEiT Transformer [45], extracts rich spatial-temporal features from each frame, enabling the model to understand and interpret complex object movements and changes in the scene.

The Pose Regressor is responsible for calculating the current frame's 6D pose, encompassing both rotation and translation vectors. Simultaneously, the Future Feature Predictor looks ahead by predicting the visual features of subsequent frames, which is crucial for maintaining temporal consistency and enhancing the robustness of pose estimations.

An innovative aspect of VideoPose is its use of past pose estimations to influence and correct the current frame's pose predictions, embodying a feedback mechanism that refines predictions over time. This method not only utilizes the spatial information present in individual frames but also dynamically incorporates historical data from the video stream, significantly boosting prediction accuracy and stability.

Even if the approach was quite outdated, it was one of the most reliable and potentially accurate with RGB images, indeed most of the other works were also relying on the depth information to regress the position. We have started our implementation from the original code, posted on GitHub: `https://github.com/ApoorvaBeedu/VideoPose`.

3.1 YOLOv8 as Object Detection Module

VideoPose assumes that the objects within the frame are already detected; hence, the 2D bounding boxes are provided as input to the model. However, there is no object detection module directly implemented. We tried three well-known approaches while developing that part:

- Mask R-CNN: Described in Section 2.1.2.

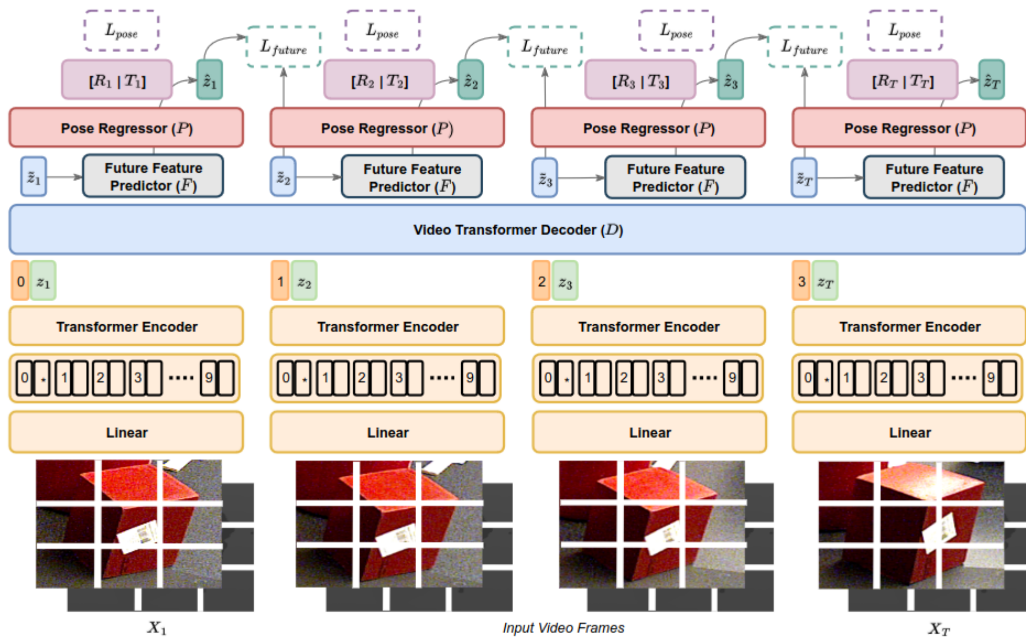


Figure 3.1. VideoPose architecture.

- MobileNet: MobileNet is a lightweight convolutional neural network architecture tailored for efficient object detection on mobile and embedded devices, developed by Howard *et al.* in 2017 [46]. MobileNet primarily utilizes depth-wise separable convolutions, which split the standard convolution into two separate layers: depth-wise convolution and point-wise convolution. This separation significantly reduces computational cost while preserving accuracy.

- YOLOv8: Described in Section 2.1.3.

Although the first two listed approaches are not as recent as YOLOv8, the reduced capabilities were good motivation for using them. However, we decided to use YOLOv8 because it provides the best trade-off between inference time and accuracy. YOLOv8 is directly available in PyTorch[1], therefore it can be directly loaded, instantiated, and trained within a few lines of code, as shown in Figure 3.2.

3.2 VideoPose Implementation

Although the released code had the possibility to be run using the flag "–no-depth", it was probably tested only using the depth information as it was not working using RGB images even including the specified flag. To fix the described issue, we have modified the model's code as it was expecting depth features even if not available. Once done, we fixed other minor bugs due to the newest library versions. The biggest problem was about adapting the model to our novel dataset (see Section 5.2 for details). We removed some post-processing functions as they were not needed for our specific task, and re-arranged the input features to be coherent with the synthetic dataset.

[1]PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is primarily used for applications such as natural language processing, computer vision, and deep learning.

```
from ultralytics import YOLO
import os

CURRENT_DIR = os.path.dirname(__file__)
CONFIG_PATH = os.path.join(CURRENT_DIR, '..', '..', 'Data', 'Configs', 'yolo_config.yml')

# Load a model
model = YOLO("yolov8n.yaml")  # build a new model from scratch

# Use the model
results = model.train(data=CONFIG_PATH, epochs=40)  # train the model
print(results)  # print results
```

Figure 3.2. Usage of YOLOv8 mode through PyTorch.

3.3 Tests and Results

This section is dedicated to the tests and results we achieved during the training and testing phase. During the preliminary training attempts, we noticed that the losses used were not converging during the epochs. This behavior was probably due to the fact that we were not using a pre-trained version of the VideoPose model. Unfortunately, this was not possible due to incompatibilities not solved by the code publisher. Indeed, the GPT version used in the decoder part was not available anymore on Hugging Face[2]. Therefore, the provided checkpoint was not working, thus making it impossible to start from a pre-trained version of VideoPose.

To check whether the problem was with the dataset or the model, we decided to download YCB-Video and run multiple trainings to reproduce the results described in the paper. Again, the same problem occurred, resulting in very bad results even on the original dataset. Due to the fact that we were not able to reproduce the results, we decided to abandon the VideoPose approach.

We attempted to train the model both using depth features and without them. However, as we can see from Figure 3.3 and Figure 3.4, as well as Figure 3.5 and Figure 3.6, the results are very similar. The graphs represent:

- ADD: It measures the average Euclidean distance between corresponding 3D object vertices predicted by the model and the ground truth 3D object vertices. It provides a quantitative measure of how closely the estimated pose matches the true pose. In this specific case, it is used as a loss instead of a score. Therefore, we expect ADD to decrease as the model improves in predicting poses. A lower ADD value, tending towards 0, indicates better performance in accurately matching the predicted poses to the ground truth.

- future_loss: This loss is used to train the future prediction heads. We expect future_loss to decrease as the model becomes better at predicting future states or poses. A decreasing future_loss suggests that the model's predictions of future states are becoming more accurate over time.

[2]Hugging Face is a company that specializes in natural language processing (NLP) technologies and develops tools and frameworks to facilitate NLP research and application development. One of its most well-known contributions is the "Transformers" library, which provides pre-trained models for various NLP tasks such as text classification, question answering, and language translation. Since vision tasks can be seen as an NLP task in which the frame patches are tokens (words), they are also used in computer vision tasks.

- rt_loss: This loss is used to train the pose regressor, measuring the error in the predicted rotation. We expect rt_loss to decrease as the model improves in predicting rotations. A lower rt_loss indicates that the model is more accurate in estimating rotational transformations.
- distance_loss: This loss is used to train the translation part of the pose regressor by indicating the error in the translation of points. We expect distance_loss to decrease as the model learns to more accurately predict translations. A lower distance_loss signifies better performance in predicting the spatial displacement of points.
- rotation_distance_loss: This loss combines both rt_loss and distance_loss. We expect rotation_distance_loss to decrease as the model improves overall in predicting both rotations and translations. A decreasing rotation_distance_loss indicates that the model is effectively learning to balance and optimize both rotational and translational aspects of pose estimation.

The two trainings were run using the following hyperparameters:

- Batch Size: 4
- Video Length: 3
- Backbone: Swin Transformer
- Epochs: 20

For training the models, an NVIDIA GeForce 4090 was utilized, and the processes ran for approximately 15 days for the model without depth features and 22 days for the one with depth features. The Swin Transformer backbone was lighter than the BEiT backbone, allowing us to increase the video length number. Therefore, at each step, the model was processing the selected frames and the previous two to take advantage of the previous predicted positions.

The ADD, in this case can also be seen as a general overview of the losses. Despite its oscillations, the average value is approximately 10.5 (considering the training phase with depth features), which is very high considering that it should tend to 0. The ADD oscillations are clearly derived from the rotation_distance_loss, which tends to suffer from the same phenomenon. The overall values of the distance_loss and rt_loss are very high at the end of the process.

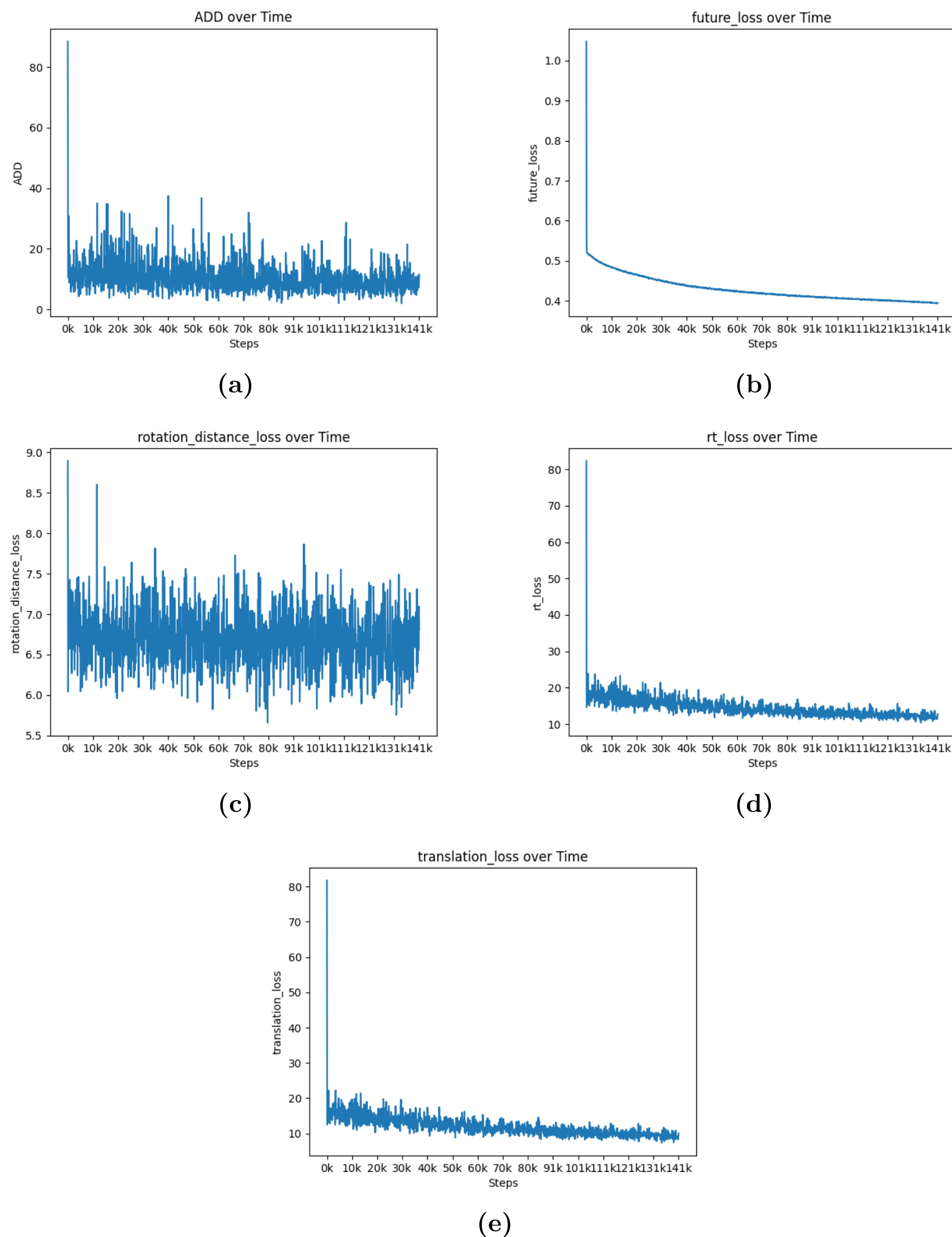


Figure 3.3. These plots represent the losses obtained during the training phase using the YCB-Video dataset and depth features. The losses shown in (a), (b), (d), and (e) appear to trend towards 0, which could misleadingly suggest that the model is learning. However, this is not the case. A key indicator of poor training is the oscillations observed in (a) and (c). While some oscillation is acceptable during the early epochs, it should reduce or disappear over time. Moreover, a closer look at the y-axes of all graphs (except for the future loss) reveals that the steps are still quite large, indicating that, even if the losses seem to be converging, the values remain too high. Finally, we can confirm the model is not learning effectively by examining the ADD graph (a), which represents the average distance between the predicted and ground truth points.

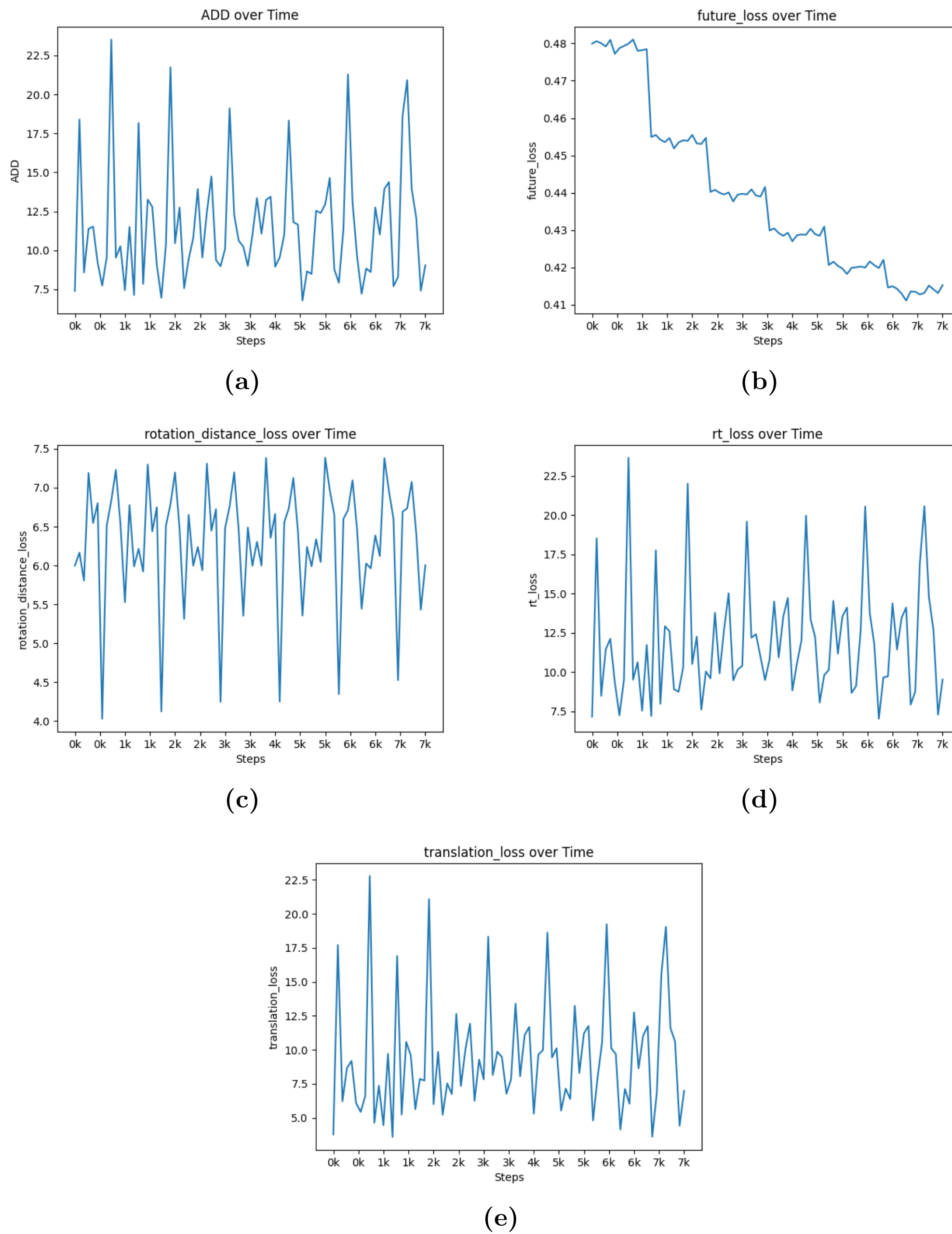


Figure 3.4. These plots represent the losses obtained during the testing phase using the YCB-Video dataset and depth features. Notably, in most graphs, we cannot even roughly estimate the trend across epochs, except for the graph concerning the future loss (b). This indicates that the model was not effectively learning to regress the object's pose. In fact, only the layer responsible for predicting the future loss was making progress.

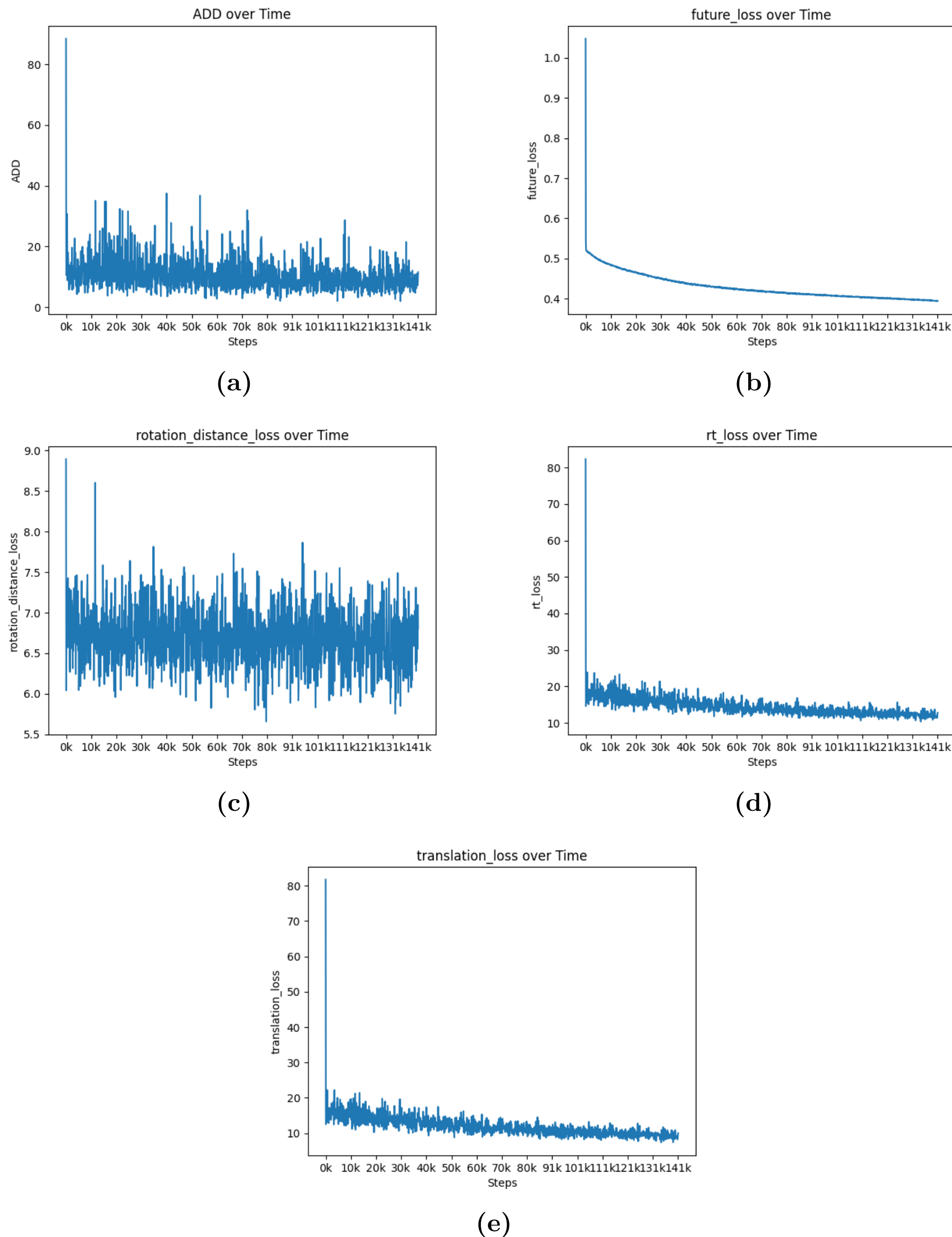


Figure 3.5. These plots represent the losses obtained during the training phase using YCB-Video dataset but without using depth features. The losses shown in (a), (b), (d), and (e) appear to trend towards 0, which could misleadingly suggest that the model is learning. However, this is not the case. A key indicator of poor training is the oscillations observed in (a) and (c). While some oscillation is acceptable during the early epochs, it should reduce or disappear over time. Moreover, a closer look at the y-axes of all graphs (except for the future loss) reveals that the steps are still quite large, indicating that, even if the losses seem to be converging, the values remain too high. Finally, we can confirm the model is not learning effectively by examining the ADD graph (a), which represents the average distance between the predicted and ground truth points.

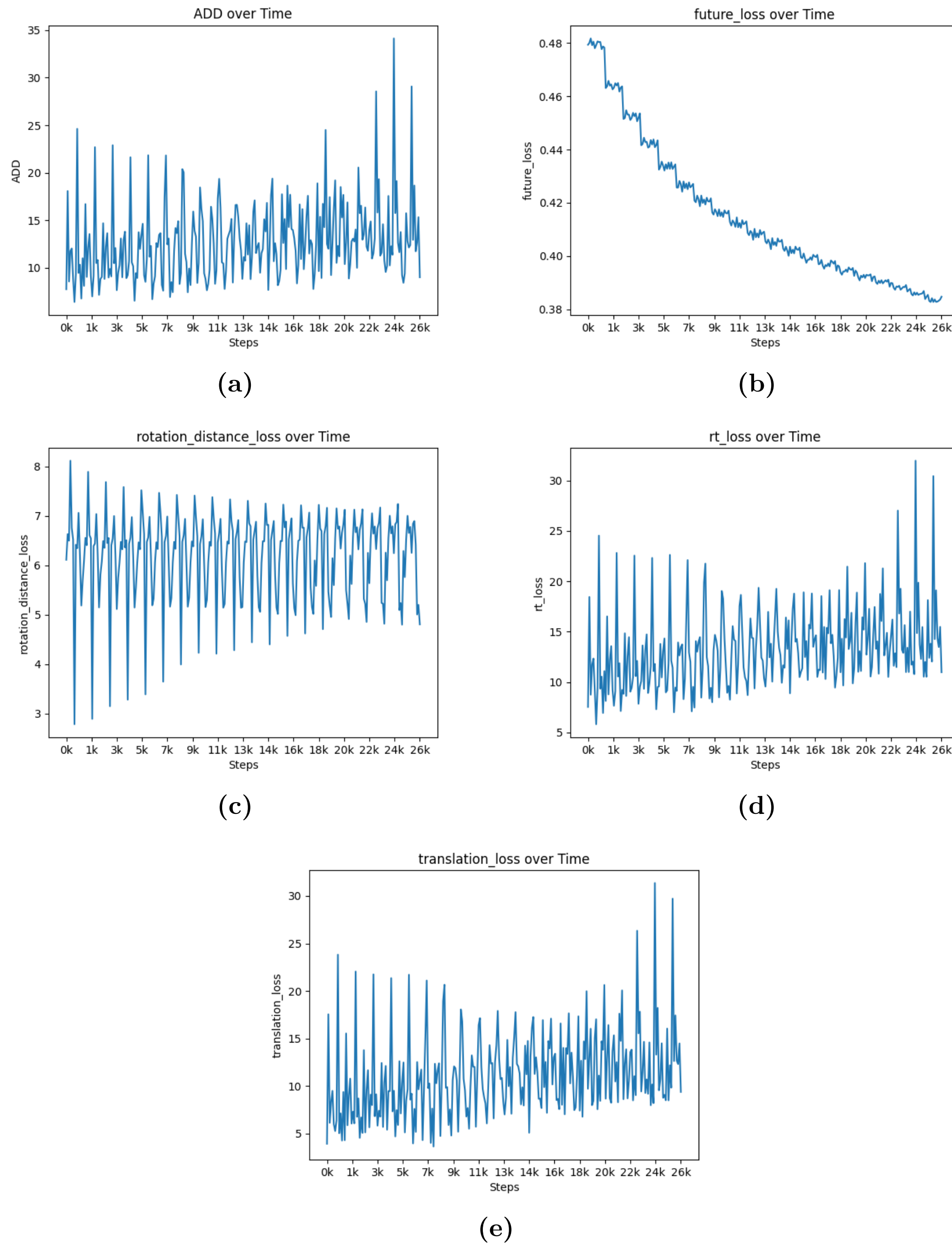


Figure 3.6. These plots represent the losses obtained during the testing phase using YCB-Video dataset but without using depth features. Notably, in most graphs, we cannot even roughly estimate the trend across epochs, except for the graph concerning the future loss (b). This indicates that the model was not effectively learning to regress the object's pose. In fact, only the layer responsible for predicting the future loss was making progress.

Chapter 4

System Architecture

4.1 Use-Cases and Goals

To provide a full and comprehensive view of the system architecture, we want to highlight the goals and use cases for which the system is designed.

4.1.1 Project Goal

As mentioned in Chapter 1, the goal was to develop a framework for Industry 4.0, ready to be deployed in an industrial environment. The framework needed to provide the ability to create a custom dataset using the user's own CAD models, ensuring compatibility with the YOLO backbone and the PoET transformer (our proposed solution), which are used for object detection and 6D pose estimation, respectively. Additionally, since the objective was to assist satellite operators during assembly phases, we aimed to develop a communication system that implements two-way communication between the model and an AR visor.

We can summarize the key contributions as follows:

1. Enabled the easy generation of synthetic images from CAD models. This generator stands out from existing solutions due to its enhanced usability and flexibility. The developed scripts require minimal configuration, allowing users to generate an entire synthetic dataset with a single terminal command.

2. Streamlined the training and evaluation of YOLO and PoET models. Similar to the synthetic data generation, the scripts allow users to quickly configure settings and achieve a well-trained, fine-tuned model. We also resolved errors and introduced new features, such as transfer learning and additional model parameters.

3. Simplified testing of both YOLO and PoET models using a webcam or visor. No public project currently offers an easy way to interface between a model and a visor. The visor used in this project was released in December 2023, and no public libraries exist for convenient access to its raw camera feeds.

4. Trained and fine-tuned both PoET and YOLO models on a given set of industrial objects. The training involved conducting 13 different experiments with 13 different configurations to compare results. The final configuration was calibrated for the specific use case, balancing computational complexity and inference-time requirements.

4.1.2 Project Use-Cases

Having defined the project's goals, we also want to briefly discuss the use cases. As mentioned in previous sections, the primary use case involves the operator assembling the satellite. Since these operations are mostly manual, it is reasonable to assume that the operator might make some errors during certain steps or may have difficulty remembering all the necessary procedures. The framework is designed to address these issues, thereby eliminating the need for a quality inspector to check for potential errors at each step. The system should provide detailed information about the current assembly step, minimizing the likelihood of incorrect interactions between the operator and the satellite.

During these operations, it is important to note that the objects will not be manipulated by the operator; thus, the camera (representing the operator) will be moving while all the objects remain stationary, taking into account the quality-check requirements. To accommodate this use case, we will test the model's performance not only analytically, providing numerical results, but also in real-world scenarios.

4.2 PoET Architecture Overview

The main part of our system architecture is PoET, it was presented in the paper: "*PoET: Pose Estimation Transformer for Single-View, Multi-Object 6D Pose Estimation*" released by Jantos *et al.* in 2023 [47]. This brilliant research has demonstrated how even a single-view RGB image could be used for performing good 6D pose estimation. The main difference with respect to VideoPose(Figure 3) stands in considering not necessarily a sequence of frames for computing the pose of an object, but leveraging the information of a single RGB image to infer the pose with the highest precision possible. The architecture shown in Figure 4.1 is mainly composed of an object detector backbone and the PoET transformer itself. The backbone not only detects and classifies the object in the frame, but it also produces intermediate feature maps. Not all detectors are able to calculate these feature maps; indeed, a pyramidal structure is needed. The authors propose three different backbones:

- **Mask R-CNN**: Already described in Section 2.1.2
- **Faster R-CNN**: Already described in Section 2.1.2
- **Scaled-YOLOv4** [48]: Scaled-YOLOv4 is an improved version of the YOLOv4 object detection model. The term "scaled" indicates that the model can efficiently scale up or down, based on the task's needs, providing a good trade-off between qualitative factors[1] and quantitative factors[2].

The best results are obtained using the Scaled-YOLOv4 backbone, which provides optimal inference time without reducing the overall precision of the network.

4.3 Scaled-YOLOv4 Backbone

Scaled-YOLOv4 is a modified version of standard YOLOv4 model presented by Wang *et al.*[48] in 2021. The main goal of this backbone is still to predict

[1] Qualitative factors may include aspects such as interpretability, ease of implementation, and robustness to various data conditions.

[2] Quantitative factors typically refer to metrics such as accuracy, speed, and computational efficiency.

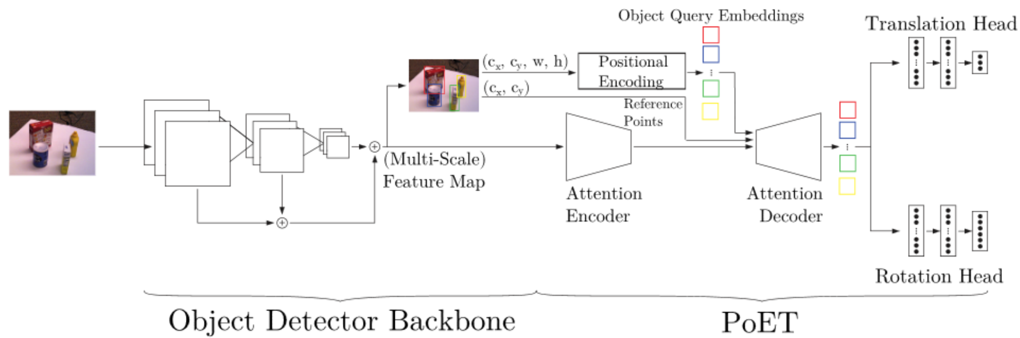


Figure 4.1. PoET architecture.

both classes and bounding boxes of the objects within a frame, but proving also intermediate multiscale feature maps. Multiscale feature maps refer to feature maps that capture information at different scales or levels of detail within an image. This allows the model to perceive objects of various sizes and shapes effectively. These are a typical output of the pyramidal-based object detector (such as YOLOv4). The "scaled" means that the model is reduced to lower the resources specifics needed to run it. This could allow the model to be executed on low-end devices with reduced computing capabilities. Typical approaches in model scaling, could lead to disadvantageous trade-offs between qualitative and quantitative factors, while the authors of the paper demonstrated that their model can reach optimal trade-offs, achieving good results even low-end devices.

4.3.1 Knowledge Basis

Before describing in detail the Scaled-YOLO paper, we want to give some more detailed overview on the topics that will be treated.

PAN Path Aggregation Module was presented for the first time in YOLOv3 [49] to address the issue of feature resolution degradation. This was a common single-stage detectors' problem derived by the feature down-sampling performed in the network. The degradation was leading to a loss of information, making less efficient and accurate the detector itself. To mitigate the problem, YOLOv3 incorporates the FPN (Section 2.1.2) and PAN. The PAN architecture enhances the feature pyramid network by introducing a path aggregation module, which efficiently combines features from different levels of the feature pyramid.

SPP Spatial Pyramid Pooling was introduced in YOLOv2 to enable the YOLO model to accept input images of arbitrary sizes while maintaining a fixed-size feature representation. The main idea behind Spatial Pyramid Pooling is to divide the final feature map generated by the convolutional layers into a grid of fixed-size regions and then apply pooling operations within each region. This allows the model to capture features at multiple scales and spatial locations, making it robust to variations in object sizes and positions within the input image.

CSPNets CSPNets [50] (Cross Stage Partial Networks), proposed by Wang *et al.* in 2019, are a particular alteration of classical CNN-based algorithms, such as ResNet, ResNeXt, and DenseNet. When CSPNets are applied to these existing

frameworks, the computational effort is reduced by 10% to 20%, outperforming the original versions in terms of both inference time and accuracy. This enhancement was made possible thanks to the analysis of existing computational bottlenecks. By analyzing CNN-based algorithms, the authors stated that at a certain layer (regardless of the architecture used), part of the gradient used in backpropagation can be considered redundant. In order to eliminate this redundancy, the proposal was to split the image into two parts: the first part is still passed through the subsequent layers, while the second part skips this step and is directly concatenated to the output of the processed part towards the end. This mechanism allows the model to run with less memory and to run faster by skipping many steps. Despite the redundancy removal, this technique is also useful for increasing accuracy.

CSP-Darknet53 CSP-Darknet53 serves as the default backbone architecture for YOLOv4 [51]. As implied by the name, it is an enhancement of Darknet53 [49] incorporating CSPNet principles. The standard Darknet53 architecture is built upon Darknet blocks, which consist of convolutional layers, batch normalization, and activation functions such as ReLU. Each Darknet block includes multiple convolutional layers followed by down-sampling layers (e.g., max-pooling or strided convolution) to progressively reduce spatial dimensions while increasing feature depth. CSPNet integration occurs at a specific layer (which may vary depending on the implementation). Following the Darknet blocks, there are additional intermediate convolutional layers for extracting high-level features. A global average pooling layer is typically applied before fully connected layers to reduce the spatial dimensions of feature maps to a single vector per feature map.

4.3.2 Proposed Models

The authors proposed three models: YOLOv4-tiny, YOLOv4-large and YOLOv4-CSP; designed respectively for low-end GPUs, high-end GPUs and general purposes.

YOLOv4-CSP The authors proposed this innovative CSP-ized version of YOLOv4. In the CSP-Darknet53 backbone, the CSP stage represents an improvement over the classical Darknet implementation only when the considered layer k satisfies $k > 1$. Therefore, the first layer of the CSP-Darknet53 was restored into the classical Darknet layer. The PAN architecture represents the neck of YOLOv4 and most of the effort has been spent to CSP-ize this part of the framework. The original PAN implemented in CSP-Darknet53, was used to integrate features given in output by different feature pyramids through a bunch of reversed Darknet residual layer[3] without shortcut connections. The authors applied the CSPNet also to this part of the network to cut down about 40% of the computation (Figure 4.2).

YOLOv4-tiny This particular version aims to drastically reduce the computational time to run the model on low-end devices. The backbone of the model is CSPPOSANet combined with PCB architecture. CSPPOSANet is an architecture based on CSP-Darknet53; it also combines PAM[4] and FPN to build reliable features. PCB (Partial Computational Block) was specifically designed to handle occluded

[3]In deep learning, residual layers are used to skip one or more subsequent layers.

[4]Position Attention Module (PAM) is designed to capture spatial dependencies and relationships between different locations within feature maps, highlighting important regions and suppressing irrelevant or redundant information.

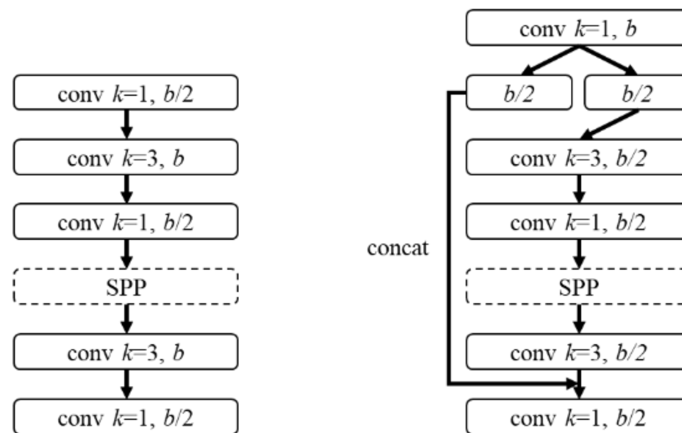


Figure 4.2. Image taken from the original Scaled-YOLOv4 paper [48]. On the left side the original PAN architecture. On the right side the CSP-ized version of PAN.

or masked regions within images. The architecture is mainly composed of three modules:

- Partial Convolutional Block (PConv): These blocks aim to exclude pixels affected by occlusion or masking from the computation, allowing the network to focus solely on the visible information.
- Partial Residual Block (PResBlock): These blocks incorporate partial convolutions within residual connections, enabling the network to learn residual features while accounting for occluded regions. By selectively processing valid pixels, PResBlock facilitates the extraction of meaningful information from occluded regions.
- Partial Upsampling Block (PUpsample): PUpsample selectively processes valid pixels to prevent the introduction of artifacts in the up-sampled feature maps.

From the tests performed on the COCO dataset, YOLOv4-tiny achieved an inference time that is eight times faster than the classical YOLOv4 model; the measured accuracy is approximately $\approx \frac{2}{3}$ of the original one. This makes the model a perfect solution for real-time object detection.

YOLOv4-large This second version is mainly designed to work with cloud GPUs (high-end devices) for very precise detections. The backbone, similarly to the classical YOLOv4, is CSP-Darknet53. Backbone's features are then passed through a FPN to scale up/down the features. What is new in this architecture, are the scale-specific detection heads. YOLOv4-large incorporates scale-specific detection heads for each of the feature scales (YOLOv4-P5, YOLOv4-P6, YOLOv4-P7). These detection heads are responsible for predicting bounding boxes and class probabilities for objects at their respective scales. After the individual detection heads produce predictions at their respective scales, the predictions are typically fused to generate the final set of detections. From the tests, the model achieved the highest accuracy score with an inference time that is about 3 times slower than the classical YOLOv4 model.

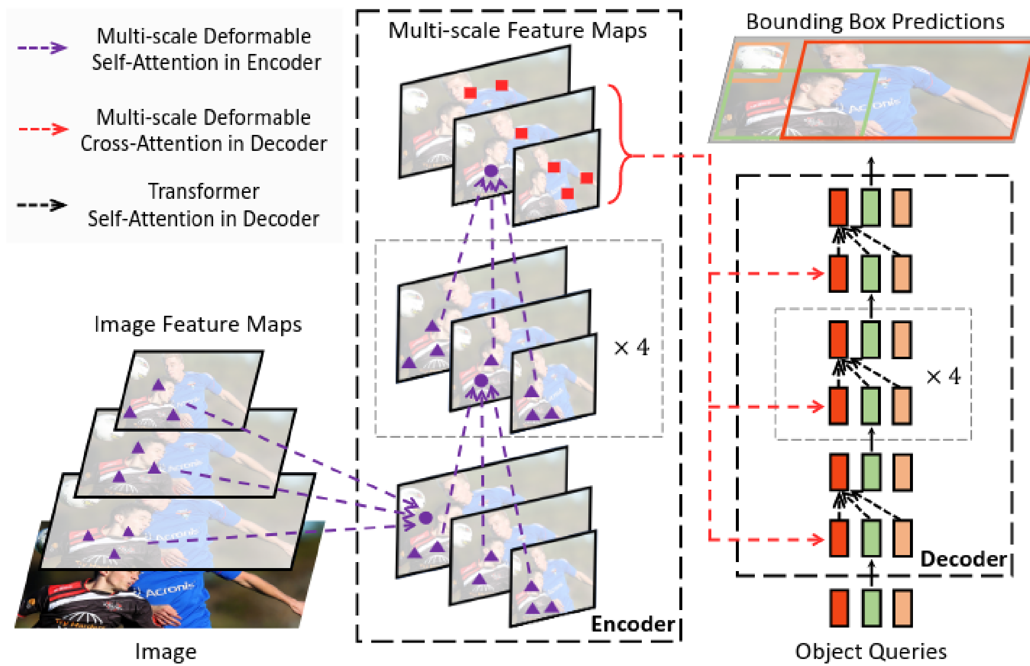


Figure 4.3. Deformable DETR, image taken from the original paper [52].

4.4 PoET Transformer

The PoET model consists of a multi-head attention-based transformer, which is a modified version of the *Deformable DETR transformer* proposed by Zhu *et al.* [52]. A Deformable transformer (Figure 4.3), in contrast with the standard DETR, can process multiscale feature maps[5]. The main idea behind using feature map refinement is to generate features that capture the global information of the frame. This typically leads to faster convergence rates than a regular transformer architecture.

Starting from the standard Deformable DETR, PoET modifies the decoder while keeping the encoder unchanged. The decoder updates aim to capture more information from the detection step. In this implementation, the decoder takes as input:

- The encoder output.
- The normalized bounding box information for each detected object (center coordinates, width, and height of the box).
- Keypoints.

In the standard Deformable DETR, bounding box information was not provided to the decoder, which instead was provided with learned object queries. Moreover, in the original Deformable DETR, keypoints were randomly sampled around some reference points. These points are learned through a fully connected layer. The

[5]Multi-scale feature maps refer to feature maps that capture information at different scales or levels of detail within an image. This allows the model to perceive objects of various sizes and shapes effectively.

authors propose to use the center coordinates of the detected bounding boxes as reference points, so the keypoints sampled around them will incorporate not only the single object information but also the global image features.

As a last step, the output features produced by the decoder are then passed as input to a rotation head and a translation head, both implemented using a Multi-Layer Perceptron (MLP). In detail:

- Translation Head: This head predicts directly the translation vector $\bar{t} = (\bar{t}_{\mathrm{x}}, \bar{t}_{\mathrm{y}}, \bar{t}_{\mathrm{z}})$, given the ground truth translation t. This head during the training phase uses the L2-loss defined as the average of the squared differences between the predicted values and the true values:

  $$L_t = ||t - \bar{t}||_2$$

  The squaring operation in the L2 loss amplifies the effect of large errors, making the model more sensitive to outliers.

- Rotation Head: This head predicts the 6D rotation vector. The authors used this representation as it does not suffer from discontinuities (while quaternions do). The head training is based upon a geodesic loss, defined as the geodesic distance between the predicted output and the true output. The geodesic distance between two points on a manifold is the length of the shortest path connecting them along the surface of the manifold:

  $$L_{rot} = \arccos \frac{1}{2}(Tr(R\bar{R}^T) - 1)$$

To combine the two losses, the authors used a weighted multitask loss:

$$L = \gamma_t L_t + \gamma_{rot} L_{rot}$$

The loss is calculated for each object and then is averaged with the resultant loss of the other objects in the batch.

4.5 Visor engine and Communication

The framework is also composed of an augmented reality core, the first idea was to use Unity to develop all the code relative to the visors. Unity is a cross-platform game engine that provides an intuitive way to develop 2D and 3D games or applications for a vast plethora of platforms such as the AR/MR/VR visors. It includes a visual editor for arranging assets, creating levels, and setting up game logic. Unity uses C# as its primary scripting language that can be also integrated with SDKs and libraries downloaded from the Asset Store.

During the preliminary phase of the project, the goal was to deploy the script as embedded[6] in the visor. We looked for existing projects about transformer deployment on visors. Nevertheless, no projects were found on that specific topic. We decided to check if at least the backbone model was deployable on the visors. In this case, we found a couple of valid projects, such as the following two:

- https://github.com/dangberg/HoloLens-YOLO-Object-Detection.
- https://github.com/doughtmw/YoloDetectionHoloLens-Unity.

[6]Embedded in this case refers to the fact that a script can run on a visor.

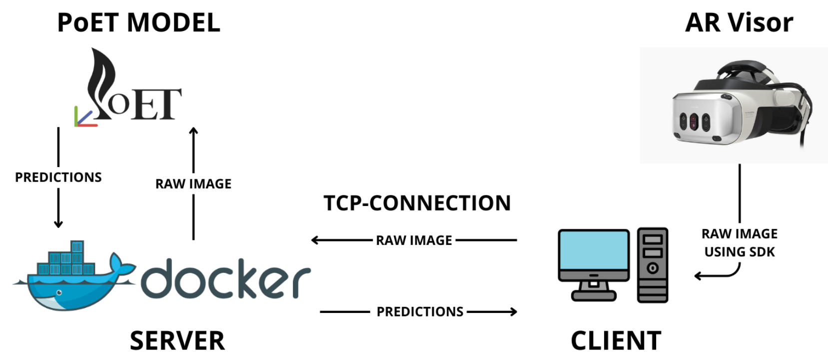


Figure 4.4. The image represents the high-level architecture in which the visor captures images through the integrated cameras and sends them to the server. The client sends these frames to the server deployed in Docker using a TCP-connection. Lastly the server sends back the predictions to the client that draws them into the operator's view.

Both of these projects are specifically designed to run on a Microsoft HoloLens. Indeed, the most important requirement to deploy a computer vision model on a visor is the ability to access the images through the API. None of the most famous visors, such as Meta or HTC ones, have the ability to access the onboard cameras, while both HoloLens and HoloLens 2 do. By going deeper into the repositories, we found out that all the projects assumed that the YOLO model was:

- Quantized: To quantize a model means that we are using techniques to reduce the capabilities needed to run the model. A typical technique is to reduce the bits used to represent the model weights. This can trivially lead to performance degradation in terms of accuracy metrics.
- Trained on a few classes (3 or 4): By reducing the number of classes, the model performance in terms of inference time can be boosted.

However, even with these assumptions, the performance in terms of FPS achieved was very poor. These projects were not suitable as starting points for our case study, as we needed to deploy not only the YOLO backbone but also the transformer. Nonetheless, this helped us understand that PoET could not be deployed onto the visor. Considering that the object detection part takes too much time to perform, this suggests that a transformer would take several seconds to output the predictions, creating a bottleneck.

The solution we devised was to abandon the idea of deploying the models as stand-alone on the visors and instead develop a client-server architecture. In this setup, the client is the visor, which can send images through a TCP socket, while the server runs on a host machine capable of receiving the images, running the model, and sending back the computed predictions to the visors (Figure 4.4).

Another important aspect we addressed was the potential for future improvements to this project. The Microsoft HoloLens is quite an old project, and as of now,

Microsoft has not announced any new visor in the HoloLens family. Therefore, we sought a more reliable solution for the future, enabling future developers to work on this project without being hindered by the outdated visor framework. After examining various industrial visors, which are not intended for the public, we found that the Varjo XR-4 visor offers an SDK for accessing the cameras. Thanks to Sapienza University and VisionLab, we were able to utilize this hardware. However, since the Varjo SDK is written in C++, it was not compatible with Unity. Consequently, we decided to abandon Unity as the engine and instead use the custom engine provided by Varjo SDK.

Chapter 5

System Implementation

5.1 Project Structure Description

We can split the project in three different parts:

1. **Synthetic Video Generation**: Described in Section 5.2.
2. **PoET**: Described in Section 5.3.
3. **Model Deployment**: Described in Section 5.4.

5.2 Synthetic Video Generation

Before developing a deep learning model or, in general, when dealing with machine learning, the first step involves discussing the needed data. A good model cannot achieve good performance without a solid dataset. In our case, the objects are covered by the Non-Disclosure Agreement (NDA)[1], therefore they are not part of a publicly available dataset, hence an ad-hoc dataset was needed. However, as discussed in Section 2.3, the most famous datasets are created by big universities and research groups. This process can take up to years to achieve a large enough labeled dataset. To drastically reduce the time and effort needed for dataset creation, we leveraged synthetic data generation techniques. Through this process we will obtain:

- GeneratedScenes: Set of labeled synthetic sequences in YCB-Video format (Section 5.2.1).
- GeneratedScenesBOP: Set of labeled synthetic sequences in BOP-Format (Section 5.2.3).
- YoloDataset: Set of labeled synthetic sequences in YOLO-Format (Section 5.2.4).

5.2.1 Blender Scripting

Blender offers a large set of Python APIs; therefore, we chose it as the core of the data rendering part. The high-level idea of the Blender script is to randomly pick a subset of objects among the available ones, then place them in a pseudo-random

[1] NDA (Non-Disclosure Agreement): A legal contract between at least two parties that outlines confidential material, knowledge, or information that the parties wish to share with one another for certain purposes but wish to restrict access to or by third parties. It is a common tool used to protect sensitive information and trade secrets.

way inside a well-defined 3D space, and lastly move the camera to render the frames. All the 3D object models were provided in two formats:

- OBJ extension: OBJ (Object) is a simple geometry file format used by Wavefront Technologies. It stores 3D models as a collection of vertices, faces, and texture coordinates. OBJ files are widely supported and can be imported into many 3D modeling software applications.
- PLY extension: PLY (Polygon File Format) is a file format used to store 3D models as a collection of polygons. It supports vertex and face data, as well as additional properties like color and normal vectors. PLY files are commonly used in computer graphics and are supported by many 3D modeling and rendering software.

In this part we used only the .obj files, as they can be loaded in Blender, while .ply are used only in the PoET model.

In order to differentiate as much as possible the generated scenes (i.e., each video is a sequence), we introduced a lot of randomized elements:

- Number, color, position, and intensity of Point Lights.
- Intensity and color of the environmental light.
- Number and position of the 3D models.
- Background texture of the scene.
- Number, duration and path of the camera movements.
- Gravity force simulation.
- Number of frames for each sequence.

Now we will discuss each point in detail to delve deeper into the project design choices. In a real-world scenario, we cannot assume that the light is uniform and always with the same intensity. By adding multiple light sources and by modifying their properties such as intensity, color and position, we modeled a general lighting condition of a room. This is fundamental also for producing noise given by the shadows and by the light that is reflected by the objects. The number of objects is also randomized to ensure that the trained model does not expect always the same number of instances in a frame. Each object cannot appear twice in a frame, this was one of the constraint imposed by the company.

In Blender, the background color can be changed; however, we cannot assign a texture to it. To overcome this issue, we decided to create a large box in which the objects can be spawned. Each panel composing the box can have a different texture, allowing us to randomly pick from a set of textures to assign to each panel (Figure 5.2).

We managed the camera movements in a way that the camera travels around the object by keeping the rotation consistent to look always in the center of the world. In detail, before starting the simulation, part of the script chooses the number of movements that the camera should make in the sequence. For each movement, the script computes the starting and target point. The target point is picked such that it lies on a 3D sphere centered at (0,0,0) with a radius $r \in [1, 2.8]$. Once the target position is fixed, the script randomly picks the duration of the movement (in seconds). This allows us to determine how many frames the camera has to reach the

```yaml
dataset_name: 'ThalesDataset'

# List of Blender files to be used for scene generation and the number of scenes to be generated for each of those
blender_files:
    '1.blend': 0
    '2.blend': 0

fps: 24  # Frames per second for animation
cam_movements_per_scene: [3, 8]  # Range of camera movements per scene (min, max)
cam_movements_duration_in_seconds: [4, 8]  # Range of duration for camera movements in seconds (min, max)


blender_settings:  # Blender settings
    range_x: [-0.3, 0.3]  # Range of x-axis coordinates for object placement
    range_y: [-0.3, 0.3]  # Range of y-axis coordinates for object placement
    range_z: [0.4, 0.9]  # Range of z-axis coordinates for object placement
    sun_energy: [0.3, 2]  # Range of environmental light intensity
    sun_color: [[0, 0.05], [0, 0.05], [0, 0.05]]  # Range of environmental light color (R, G, B)
    max_lamp_num: 3  # Maximum number of lamps to be added to the scene
    lamp_brightness: [0.1, 1]  # Range of lamp brightness
    lamp_colors: [[0.5, 1], [0.5, 1], [0.5, 1]]  # Range of lamp colors (R, G, B)
    lamp_pos_range: [[-1.5, 1.5], [0.5, 1.5], [-1.5, 1.5]]  # Range of lamp positions (X, Y, Z)


camera_settings:  # Intrinsic camera parameters
    height: 480
    width: 640
    focalX: 1.066778000000000020e+03
    focalY: 1.067487000000000080e+03
    centerX: 3.129868999999999915e+02
    centerY: 2.413109000000000037e+02
```

Figure 5.1. YAML configuration file, through these parameters we can modify each single aspect of the scene generation without the need of changing the code.

destination starting from its current position. The route of the camera is not linear; instead, it follows a curved trajectory to guarantee smooth movement along the path. This movement is implemented through a Bézier curve, a parametric curve used in computer graphics and animation to create smooth and elegant shapes and trajectories. It is defined by a set of control points that influence the shape of the curve:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i$$

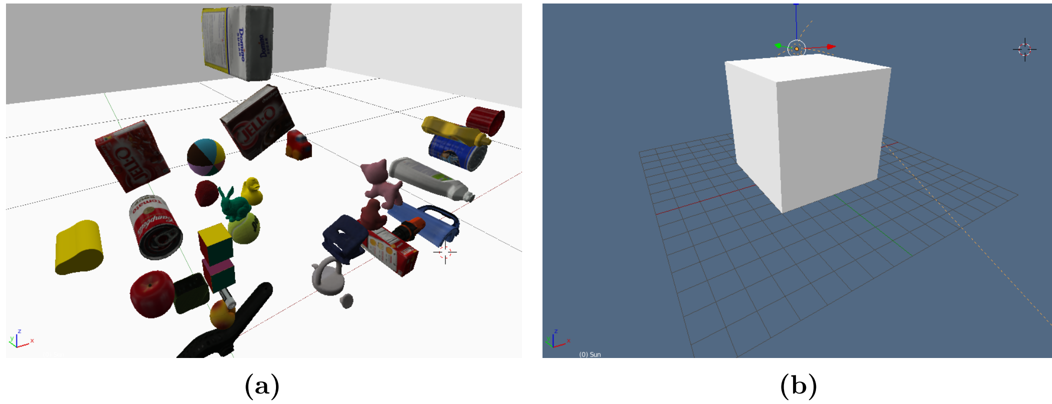
For each frame of the movement sequence, the script computes a keyframe, saving both rotation and position of the camera. During rendering, by simply setting the keyframe, the camera will be moved and rotated accordingly to the stored values.

We decided not only to (pseudo) randomly place the objects in the box but also to apply a gravity force with an intensity $f \in [-2, 2]$. This force is applied only for a few seconds, allowing the objects to "interact" with each other and creating more realistic poses in the scene.

The last randomized parameter is the number of frames per sequence, this number is picked in a range specified by the user.

One of the goal of this script was also to make it fully configurable without changing anything in the code, we implemented a configuration YAML file in which every single aspect of the generation can be changed (Figure 5.1).

The most important parameter is the number of scenes to be generated for each blend file. In our experiments, we implemented two different Blender scenes (i.e., blend files). The first one is already populated with other object models (retrieved from the YCB-Video dataset), while the second one is empty. The main goal of the first blend file is to include objects that will not be labeled in the dataset to produce occlusion (Figure 5.2).



(a) (b)

Figure 5.2. On the left side: First blend file with preloaded objects. The background texture is not set as it will be dynamically loaded in the script. On the right side: Box in the blend file containing the objects.

Labeling For producing a dataset, we need a label file associated to each frame representing the needed information about the whole image. As described in Section 2.3, most of the state-of-the-art approaches, tend to test their models on at least two or three datasets. The most widely used is YCB-Video dataset, therefore we decided to adopt its labeling format. Online you can find mainly two YCB versions, the original version and the one provided in BOP format. In this specific step, we chose the classical labeling format with the following structure:

```
Dataset/
├── 0000/
│   ├── 000000-color.png
│   ├── 000000-depth.png
│   ├── 000000-seg.png
│   ├── 000000-meta.npy
│   ├── 000001-color.png
│   ├── 000001-depth.png
│   ├── 000001-seg.png
│   ├── 000001-meta.npy
│   ├── ...
│   ├── xxxxxx-color.png
│   ├── xxxxxx-depth.png
│   ├── xxxxxx-seg.png
│   └── xxxxxx-meta.npy
├── 0001/
│   ├── ...
│   ├── yyyyyy-color.png
│   ├── yyyyyy-depth.png
│   ├── yyyyyy-seg.png
│   └── yyyyyy-meta.npy
├── ...
└── zzzz/
```

Each folder under the Dataset/ folder (i.e. 0000/, ..., zzzz/) represent a video sequence and contains four files for each frame:

- xxxxxx-color.png: RGB frame.
- xxxxxx-depth.png: Depth map.
- xxxxxx-seg.png: Semantic segmentation where each object is segmented with a different color.
- xxxxxx-meta.npy: NumPy[2] file containing all the frame's labels. The structure of each NumPy file is the following:
  - cls_indexes: Array containing the IDs of the object in the frame.
  - poses: 4x4 matrix specifying the 6DoF pose of the objects (-th matrix corresponds to the 6Dposes of the -th model in cls_indexes).
  - blendercam_in_world: 4x4 matrix specifying the 6DoF pose of the camera.
  - intrinsic_matrix: Matrix specifying the camera intrinsic parameters.

By examining each pose of a certain object in a sequence, we notice that it remains consistent across frames. While this might seem counterintuitive, it's important to consider how the frames are generated: the object's pose remains unchanged during sequence rendering, while it's the camera position that is updated. In pose estimation tasks, we cannot directly regress the real object pose; instead, we regress the object pose relative to the camera. In other words, we assume that the camera is static, representing the center of the world, while the entire external environment moves. Given the real object pose, and the camera pose, we can simply apply geometric transformations to obtain the 4x4 matrix representing the object pose relative to the camera's view. Given the 4x4 6-DoF matrix representing the camera pose C and the 4x4 6-DoF matrix representing the object pose O (representing the real-world pose of the object), the object poses relative to the camera's view O_{relative} can be obtained using the following operations:

$$\begin{aligned} O_{\text{relative}} &= C^{-1} \times O \\ &= \begin{bmatrix} R_C & t_C \\ 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} R_O & t_O \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_C^T & -R_C^T \cdot t_C \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} R_O & t_O \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_C^T \cdot R_O & R_C^T \cdot (t_O - t_C) \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{5.1}$$

Here, R_C and t_C are the rotation and translation components of the camera pose matrix C respectively, and R_O and t_O are the rotation and translation components of the object pose matrix O respectively.

[2]NumPy is a fundamental package for scientific computing with Python. It provides support for large, multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy is widely used in various fields such as physics, engineering, finance, machine learning, and more.

(a) (b)

Figure 5.3. Two examples of generated frames, as we can see they have different background textures and different light settings.

5.2.2 Bounding Box Generation

In Section 2.2, we stated that to perform accurate pose estimation, we typically rely on accurate object detection. To train an object detector, we need ad-hoc labeled data. Generally, what we need in terms of labeling are the bounding boxes contained within a frame and the associated object classes. In the previous Section 5.2.1, we discussed how the synthetic dataset is created. However, during the labeling phase, we didn't include any information about the bounding boxes in the frame. Indeed, we developed a different script to create these additional labels. Typically, these bounding boxes are not needed by pose regression algorithms, but only by the object detection algorithms, therefore we decided to split the labels into two groups:

- Poses labels (the .npy files generated during the frame rendering).
- Boxes labels.

One of the main goals of this synthetic video generation part was not only to provide a solution for creating the needed dataset for our purposes, but also to offer a tool for other developers. Therefore, even if the approach described in the next sections relies on 2D bounding boxes to detect the object, we decided to also generate 3D bounding box labels as all the needed information to compute them were already available.

2D Bounding Boxes Generation

To understand the steps described below, we must introduce the concept of an object mesh. A 3D model of an object, also known as a mesh, can be seen as a set of (typically millions) 3D points describing the shape of that model. The more complex the mesh, the higher the number of points needed to describe it. Consider a 2D world: when representing a square, we need only 4 points as it is a very simple shape. On the other hand, if we want to draw a circle, using only 10 points will make it appear edgy; the more points we add, the smoother the circle will appear. The same concept can be extended into the 3D world.

The high-level idea behind the 2D box generation is simple. Given the mesh of an object, its 4x4 6DoF matrix, and the 4x4 6DoF matrix of the camera, we can first apply the pose of the object to the mesh, hence obtaining the list of 3D points

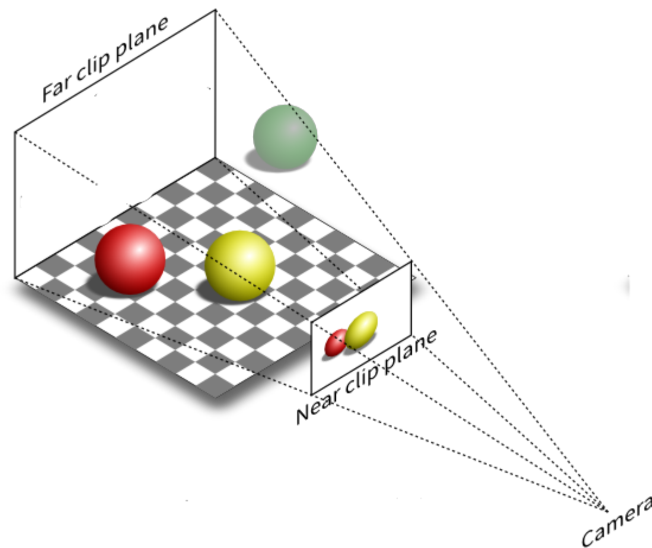


Figure 5.4. 3D object projection on a 2D plane with respect to the camera view.

composing the object and their position in the frame. Then, using Equation 5.1, we can obtain the position of these 3D points with respect to the camera; now, they can be projected onto the 2D plane. By projecting them, we obtain the 2D position of those points in the frame. The last step to obtain the bounding box of that object is trivial: using a simple min-max function, we get the up-left most and down-right most corners of the box. The other two corners can be easily computed using these two known corners.

This process is repeated for each object in each frame of the sequence, saving the known corners in a .txt file (one for each frame).

Algorithm 1 2D Box Generation

Require: Object mesh M, object pose matrix P_{object}, camera pose matrix P_{camera}
Ensure: Bounding box coordinates B
Apply object pose to mesh: $M' \leftarrow P_{\text{object}} \times M$
Convert 3D points to camera frame: $M'' \leftarrow P_{\text{camera}}^{-1} \times M'$
Project 3D points onto 2D plane: $P \leftarrow \text{project}(M'')$
Compute bounding box coordinates: $B \leftarrow \text{compute_bounding_box}(P)$
return 2D bbox vertices B

Algorithm 2 Compute 2D Bounding Box

```
function COMPUTE_2D_BOUNDING_BOX(P)
    x_min ← min(P_x)          ▷ Minimum x-coordinate
    x_max ← max(P_x)          ▷ Maximum x-coordinate
    y_min ← min(P_y)          ▷ Minimum y-coordinate
    y_max ← max(P_y)          ▷ Maximum y-coordinate
    return Bounding box coordinates: (x_min, y_min), (x_max, y_max)
end function
```

3D Bounding Boxes Generation

The 3D box generation is mostly similar to the process described for the 2D box generation. Again, starting from the object mesh and its 6DoF pose, we can compute the mesh points' 3D positions. However, before projecting them onto the camera plane, we have first to compute the 8 vertices of the 3D bounding box; otherwise, the 2D projection will result in losing some information needed to compute the 3D vertices. We can consider the 3D bounding box as two 2D boxes with the corners joined. Therefore, similarly to what we did in Algorithm 2 for computing the 2D boxes, we can calculate the following points describing the first of the two boxes composing the 3D final box:

- $x_{\min}, y_{\min}, z_{\min}$
- $x_{\min}, y_{\max}, z_{\min}$
- $x_{\max}, y_{\min}, z_{\min}$
- $x_{\max}, y_{\max}, z_{\min}$

The second box is quite trivial:

- $x_{\min}, y_{\min}, z_{\max}$
- $x_{\min}, y_{\max}, z_{\max}$
- $x_{\max}, y_{\min}, z_{\max}$
- $x_{\max}, y_{\max}, z_{\max}$

These 8 points represent the 8 vertices describing the 3D bounding box of an object. Additionally, we also provide the projected 3D bounding box, simply obtained by using Equation 5.1 to convert the points' coordinates into the camera perspective and projecting them onto the 2D plane.

Algorithm 3 3D Box Generation

Require: Object mesh M, object pose matrix P_{object}, camera pose matrix P_{camera}
Ensure: 3D bounding box coordinates B
  Apply object pose to mesh: $M' \leftarrow P_{\text{object}} \times M$
  Get 3D bounding box vertices: $V \leftarrow \text{get_3d_bbox_vertices}(M')$
  Convert 3D points to camera frame: $V' \leftarrow P_{\text{camera}}^{-1} \times V$
  Project 3D bbox vertices onto 2D camera plane: $V'' \leftarrow \text{project}(V')$
  return 3D bbox vertices V, projected 3D bbox vertices V''

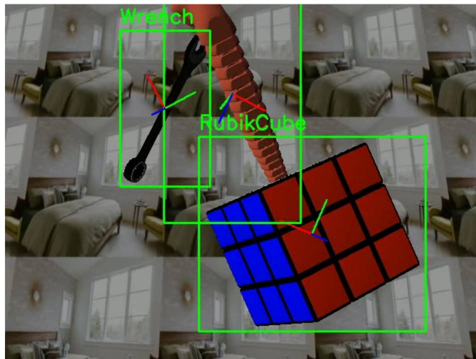
5.2.3 YCB-Video BOP Format Conversion

In Section 5.2.1, we stated that during the labeling phase we adopted the original YCB-Video labeling format for producing our custom labels. This synthetic video generation part has been developed in parallel with the pose detection algorithm. The first approach we used required a YCB-like dataset, while the second approach (Section 5.3) used the BOP format. To be more flexible, we developed a script to convert the standard YCB-Video format into the BOP-compliant one, offering other

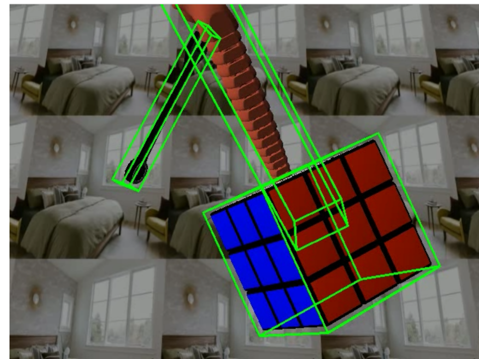
Algorithm 4 Compute 3D Bounding Box

function COMPUTE_3D_BOUNDING_BOX(P)
- $x_{\min} \leftarrow \min(P_x)$ ▷ Minimum x-coordinate
- $x_{\max} \leftarrow \max(P_x)$ ▷ Maximum x-coordinate
- $y_{\min} \leftarrow \min(P_y)$ ▷ Minimum y-coordinate
- $y_{\max} \leftarrow \max(P_y)$ ▷ Maximum y-coordinate
- $z_{\min} \leftarrow \min(P_z)$ ▷ Minimum z-coordinate
- $z_{\max} \leftarrow \max(P_z)$ ▷ Maximum z-coordinate
- **return** {
  - $(x_{\min}, y_{\min}, z_{\min})$,
  - $(x_{\min}, y_{\max}, z_{\min})$,
  - $(x_{\max}, y_{\min}, z_{\min})$,
  - $(x_{\max}, y_{\max}, z_{\min})$,
  - $(x_{\min}, y_{\min}, z_{\max})$,
  - $(x_{\min}, y_{\max}, z_{\max})$,
  - $(x_{\max}, y_{\min}, z_{\max})$,
  - $(x_{\max}, y_{\max}, z_{\max})$
- }

end function



(a) Frame of a sequence with drawn poses and 2D bounding boxes of the objects in the scene.



(b) Frame of a sequence with drawn 3D bounding boxes of the objects in the scene.

Figure 5.5

researchers the possibility to choose between the two provided formats. The BOP format requires the following structure:

```
Dataset/
 ├── train/
 │    ├── 0000/
 │    │    ├── depth/
 │    │    │    ├── 0000001.png
 │    │    │    ├── 0000002.png
 │    │    │    ├── ...
 │    │    │    └── xxxxxxx.png
 │    │    ├── rgb/
 │    │    │    ├── 0000001.png
 │    │    │    ├── 0000002.png
 │    │    │    ├── ...
 │    │    │    └── xxxxxxx.png
 │    │    ├── mask/
 │    │    │    ├── 0000001.png
 │    │    │    ├── 0000002.png
 │    │    │    ├── ...
 │    │    │    └── xxxxxxx.png
 │    │    ├── scene_camera.json
 │    │    ├── scene_gt_info.json
 │    │    └── scene_gt.json
 │    ├── 0001/
 │    ├── 0002/
 │    ├── ...
 │    └── zzz0/
 └── test/
      ├── zzz1/
      │    ├── depth/
      │    ├── rgb/
      │    ├── mask/
      │    ├── scene_camera.json
      │    ├── scene_gt_info.json
      │    └── scene_gt.json
      ├── zzz2/
      ├── ...
      └── yyyy/
```

As we can see, files are split into training and testing sets (respectively in train/ and test/ folders), each of which contains folders representing the video sequences. In each sequence folder, we can find:

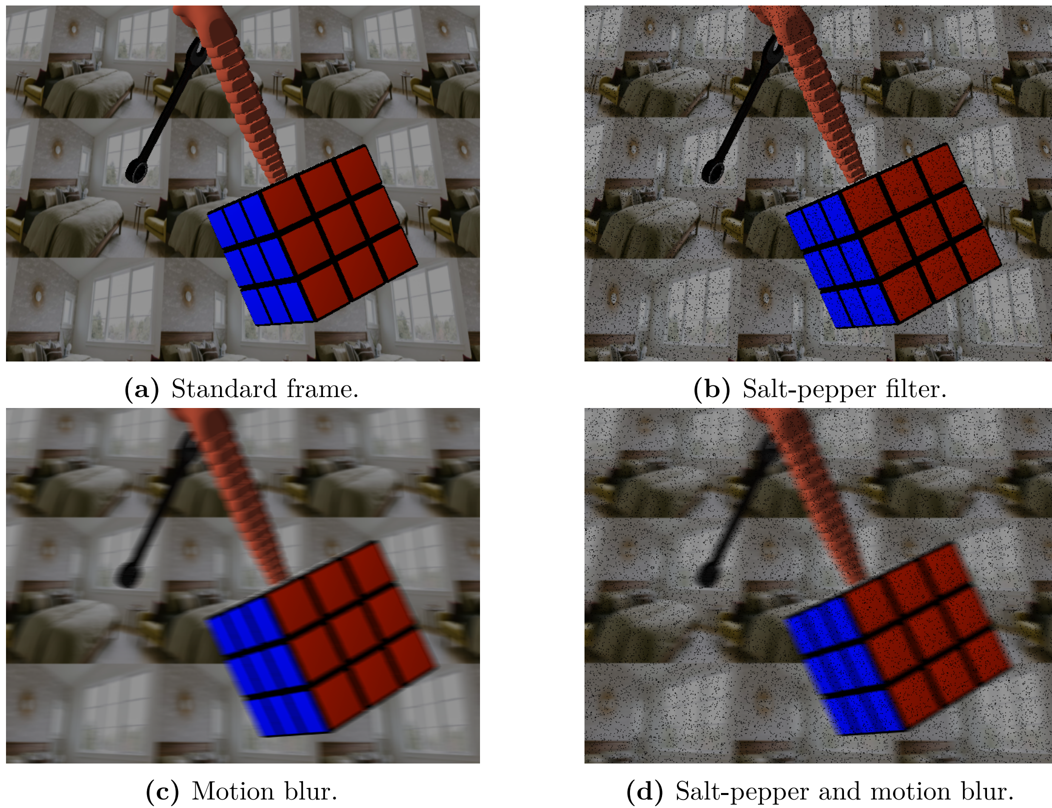
- depth/ : Folder containing all the depth maps of the sequence.
- rgb/ : Folder containing all the RGB frames of the sequence.
- mask/ : Folder containing all the semantic segmentation maps of the sequence.

- scene_camera.json: JSON file containing all the information of the camera position and intrinsic across the whole sequence.
- scene_gt_info.json: JSON file containing all the information of each object's bounding box of each object in each sequence.
- scene_pose.json: JSON file containing all the information of each object's pose of each object in each sequence.

Once we converted the dataset from YCB format to the BOP compliant one, we also added a data distortion phase. In this part, we randomly modified images in both the testing and training sets to make them more realistic. This step aims to enhance the training of the model to prevent the pose regression model from being unable to transfer the acquired knowledge from the synthetic data to the real one during the inference phase. When a deep learning model is trained and tested only using synthetic data, there is a high probability that if it is used in inference mode[3] with real data, we could have very poor results, even if the metrics on the synthetic data are up to the state-of-the-art level.

To address this issue when dealing with synthetic data, we add some "noise" to these data to make them more "realistic". A typical strategy is to apply rotations, image clippings, blur, and salt-pepper noise. However, while rotating and clipping images are not useful for our dataset because the camera moves and rotates randomly, hence there is no need to flip the images or apply similar operations, applying some motion blur and salt-pepper filter could help our dataset become more noisy. In particular, we added some motion blur to the image rather than using classical blurring filters, as it is much more realistic to simulate the blur obtained by moving a real camera (Figure 5.6).

[3]Inference mode refers to the operational state of a trained model where it processes new, unseen data to make predictions or classifications.



(a) Standard frame.

(b) Salt-pepper filter.

(c) Motion blur.

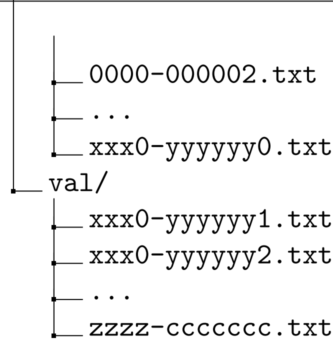
(d) Salt-pepper and motion blur.

Figure 5.6

5.2.4 YOLO Format Conversion

The most widely used as object detection algorithm is YOLO. This model requires a well-known dataset structure to work, therefore we also had to format the dataset structure to create a YOLO compliant dataset. The overall file arrangement is the following:

```
Dataset/
├── images/
│   ├── train/
│   │   ├── 0000-000001.png
│   │   ├── 0000-000002.png
│   │   ├── ...
│   │   └── xxx0-yyyyyy0.png
│   └── val/
│       ├── xxx0-yyyyyy1.png
│       ├── xxx0-yyyyyy2.png
│       ├── ...
│       └── zzzz-ccccccc.png
└── labels/
    └── train/
        └── 0000-000001.txt
```



Trivially, for each frame .png we have an associated .txt labeling file containing the IDs of the models in the frame and the relative bounding boxes. However, in YOLO the bounding boxes are not represented as in YCB or BOP formats (hence using two opposite corners of the bounding box) but through the notation: (c_x, c_y, w, h), where:

- (c_x, c_y): Center coordinates of the bounding box.
- h: Height of the bounding box.
- w: Width of the bounding box.

5.3 PoET Implementation

For the paper implementation we started from the public PoET's GitHub repository: `https://github.com/aau-cns/poet`. The repository uses Docker[4] to run the project, without needs of setting up external libraries or dependencies. In particular, the docker image settings are:

- Ubuntu 20.04
- CUDA 11.4
- Python 3.8.8
- PyTorch 1.9
- Standard packages: NumPy, SciPy, cv2, Cython
- Non-standard packages: mish-cuda, deformable_attention (for the Deformable DETR code).

5.3.1 Scaled-YOLOv4 Implementation

In the PoET GitHub repository, Mask-RCNN and Faster-RCNN backbones are provided by default, while the Scaled-YOLOv4 version is not included. However, the authors provide an additional link to retrieve the Scaled-YOLOv4 implementation: `https://github.com/aau-cns/yolov4`. Initially, our work involved integrating the backbone code into the PoET repository.

The backbone cannot be trained with the PoET transformer itself; it must be trained separately. For the training step, we had two choices:

[4] Docker is a platform that enables developers to build, package, distribute, and run applications in containers. These containers encapsulate all the dependencies required for an application, including libraries, runtime environments, and system tools, ensuring consistency and portability across different computing environments.

- Train the model from scratch: Training a model without starting from a pre-trained model can often be very time and resource-intensive. However, it also has many advantages, as not all pre-trained models can be easily adapted to work with a different dataset.
- Train the model using transfer-learning techniques: This technique involves reusing or adapting a model trained on one task as the starting point for a related task. Instead of starting the learning process from scratch, transfer learning leverages knowledge gained from solving one problem to solve a different but related problem more efficiently. Starting with a pre-trained model can save a lot of time, as we only need to fine-tune a subset of the model's layers for the new dataset. However, we need to adjust the model configurations when dealing with a different dataset than the one the model was initially trained on. This technique can also drastically reduce the number of data instances required to perform a good training step.

In this specific context, we decided to perform transfer learning, since the training-from-scratch technique would have taken more time than the chosen method and required too many video sequences, also probably leading to worse performance.

To perform transfer-learning, we mainly needed two files:

- Model configuration file (.cfg file)
- Model weights (.pt file)

Unfortunately, these files were not publicly available, so we contacted the repository's owner, who provided us with the necessary files.

Scaled-YOLOv4 Configuration File

The model configuration file contains specifications about all the layers that compose the model (Figure 5.7). The layers are specified with the syntax "[layer_name]". We can directly modify values in the file to adjust an existing model to our new needs. The general rule when dealing with transfer learning (always considering a configuration file generated for a different dataset) and YOLO configuration file, is to modify two attributes:

- "classes": In the .cfg file we will find more than one layer named "[yolo]", in each of those we have to modify such attribute indicating the number of classes we have to predict.
- "filters": Each "[yolo]" layer is preceded by a "[convolutional]" layer. In this layer, we have to modify the filters attribute with the value x that satisfies the following equation:

$$x = (\text{num_classes} + 5) * 3 \tag{5.2}$$

Scaled-YOLOv4 Freezing

The model weights represent the parameters learned during the training process. Trivially, the parameters that are valid for a certain dataset, with a high probability, will not be valid for a different dataset. However, we can gain advantages from those with the transfer learning technique. Each model layer has a set of parameters used for producing a certain output that will be processed by the subsequent layer.

```
size=3
stride=1
pad=1
filters=1024
activation=mish

[convolutional]
size=1
stride=1
pad=1
filters=78
activation=linear


[yolo]
mask = 6,7,8
anchors = 12, 16, 19, 36, 40, 28, 36, 75, 76, 55, 72, 146, 142, 110, 192, 243, 459, 401
classes=21
num=9
jitter=.3
ignore_thresh = .7
truth_thresh = 1
random=1
scale_x_y = 1.05
iou_thresh=0.213
cls_normalizer=1.0
iou_normalizer=0.07
iou_loss=ciou
nms_kind=greedynms
beta_nms=0.6
```

Figure 5.7. Example of two layers in the Scaled-YOLOv4 configuration file.

Typically, the first layers in an architecture are not specialized in the final task (in this case, object classification and detection), but they are used to capturing high-level features such as patterns, corners, edges, and simple shapes. Indeed, two models (with the same architecture) trained for the same task on different datasets could likely have similar parameters in the higher layers. In such a case, we can apply the freezing-layers technique; this method involves "freezing" a certain amount of layers. Namely, during the training backward pass, the gradients associated with these layers will not be updated. This ensures that only the last layers are trained with the new dataset, hence reducing the overall training time needed to learn new parameters.

In PyTorch, there are no functions that directly freeze a certain amount of layers in the model, hence we introduced the code to freeze a given percentage of layers as shown in Figure 5.8.

YOLO Bash Training Script

We also addressed the lack of usability in the provided code. The Python code was difficult to use and read, which led to a significant waste of time when trying to find or modify training parameters. To improve this, we removed all hard-coded variables and moved them into a YAML file used to specify all model parameters (e.g., number of classes, layers, etc.). This change allowed us to create a bash script that performs all the preliminary checks before launching the training with the parameters specified in the YAML file (Figure 5.9).

```python
# Freezing layers
total_params = len(list(model.parameters()))
freeze_until = int((opt.to_freeze / 100) * total_params)  # Calculate the number to freeze
print('Freezing the first {} layers'.format(opt.to_freeze))
for i, (name, param) in enumerate(model.named_parameters()):
    if i < freeze_until:
        print('Freezing layer {}'.format(name))
        param.requires_grad = False
```

Figure 5.8. This code demonstrates how we implemented the freezing step before starting training in YOLO. As shown, given a percentage, the loop disables gradient tracking for the specified layers to prevent their update during training. This speeds up the training process by reducing the number of parameters tracked during backpropagation, albeit at the cost of some overall precision.

```bash
# Ensure mandatory parameters are specified
if [[ -z "$dataset_name" ]]; then
    echo "Error: --dataset_name is required."
    usage
fi

if [[ -z "$gpu_mem_size" ]]; then
    echo "Error: --gpu_mem_size is required."
    usage
fi

# Check if dataset folder exists
if [[ ! -d "$(pwd)/../../Datasets/$dataset_name/YoloDatasetV2" ]]; then
    echo "Error: Dataset directory ../../Datasets/$dataset_name/YoloDatasetV2 does not exist."
    exit 1
fi

# Run Docker container
sudo docker run --entrypoint= -v $(pwd)/../../CustomPoET:/opt/project \
    -v $(pwd)/../../Datasets/$dataset_name/YoloDatasetV2/:/YoloDataset \
    --shm-size=${gpu_mem_size}g --rm --gpus all aaucns/poet:latest python \
    -u /opt/project/models/yolov4/yolo/train.py \
    --data="/opt/project/models/yolov4/yolo/data/$dataset_name.yaml" \
    --names="/opt/project/models/yolov4/yolo/data/$dataset_name.names"
```

Figure 5.9. YOLO bash training script. Note that we developed the script to be easily used without the need to run all commands manually. This is part of the usability refactoring that we applied to the existing project.

5.3.2 PoET Transformer Implementation

During the initialization of the transformer, certain necessary parameters must be specified, including:

- Backbone instance
- Deformable Transformer
- Number of encoding layers
- Number of decoding layers
- Number of classes
- Rotation representation
- Bounding box mode
- Max number of queries

As specified in Section 4.4, PoET consists of two main building blocks: the backbone and the basic transformer. In this context, the backbone is the Scaled-YOLOv4 implementation. As mentioned in preceding sections, the backbone cannot be trained simultaneously with the PoET transformer. Therefore, an already trained and fine-tuned backbone instance must be provided during the transformer's training. However, the model can also be trained using ground-truth data by specifying it through the parameter "bounding box mode". The advantages and disadvantages of training the model in ground-truth mode versus prediction mode will be discussed in Chapter 6.

In the dedicated section, it was mentioned that the PoET transformer's main building block is the Deformable DETR. Thus, when instantiating the model, a Deformable DETR instance must be created first. However, Deformable DETR is not provided within the canonical PyTorch library. Instead, it is installed from the original Deformable DETR GitHub repository: `https://github.com/fundamentalvision/Deformable-DETR`.

The number of encoding and decoding layers specified in the PoET input represents the encoding and decoding layers in the Deformable DETR instance. Higher values for these parameters result in a more complex transformer.

Lastly, the number of queries represents the maximum number of target-object instances that can be found in a single frame.

In the original code, there was the possibility to start the training either from scratch or by using a pre-trained model as a basis. Therefore, we first decided to adopt a similar solution to the one proposed in Chapter 5.3.1. For Scaled-YOLOv4, we froze the first layers to focus all the training efforts on the last layers. Nevertheless, the PoET transformer is more complicated than the YOLO architecture, as it is composed of encoding layers, decoding layers, and two MLPs (translation and rotation heads). Therefore, it is much more challenging to understand which layers should be frozen and how. Moreover, in the PoET checkpoint, there are also the optimizer and learning rate scheduler statuses saved. For all these reasons, we decided to train from scratch the entire PoET transformer.

Backbone Empty Predictions Problem Fix

During the preliminary tests, we noticed that the transformer was continuously crashing during training. Upon further investigation, we determined that the model always expected at least one prediction from the backbone. While this was not an issue from the authors' perspective, in our use case, we cannot guarantee that an object is always present in the scene. To address this, the authors introduced a pre-processing step that removes all images without objects from the dataset. However, this pre-processing step was still insufficient to prevent the transformer from crashing due to missing predictions. Indeed, when training the model with ground-truth labels, the pre-processing step worked as intended. But when training the model in backbone mode, the same error persisted. Even when an object was present in the frame, the transformer would crash if the backbone failed to detect it.

Since we wanted to train the model on empty frames, and the solution proposed by the authors was not working properly, we removed the pre-processing step to ensure that images without objects were included. To resolve the missing predictions error, we modified the existing code so that the transformer first checks for any predictions. For each missing object query (i.e., PoET always expects a fixed number of objects in the frame, known as object queries), we handle the discrepancy by adjusting the predictions. If there are more predicted objects than object queries, we simply remove the excess predictions. Conversely, if there are fewer predictions, we fill the unassigned object queries by populating the tensors with a value of -1. These dummy tensors are then pruned during the loss calculation. If no predictions are retrieved from the backbone, we will end up with all dummy tensors, resulting in a $\pm\infty$ value. To handle this, we introduced a memory mechanism where we save the previous loss. Whenever a $\pm\infty$ value is encountered, we simply return the loss from the previous step, effectively ignoring the current batch during the backpropagation step.

Through these simple yet effective steps, we solved the problem and improved the model's generalization by allowing it to be trained with examples that include empty frames.

5.4 Model Deployment

In this section, we will describe the implementation of the inference architecture outlined in Section 4.5.

5.4.1 YOLO Inference Script

We proceeded to create a server capable of running real-time predictions using only the backbone model. This script was not necessary for the one described in Section 5.4.2. However, since we obtained a well-performing YOLO checkpoint some weeks before the PoET checkpoint, and we wanted to showcase the achieved results, we created the following script to test the model using a simple webcam.

We first developed the server. In the Scaled-YOLOv4 repository, there was already a script capable of taking images from a webcam. Therefore, the server first waits for an incoming connection from a client, then reads the images through the webcam and feeds them into the model. YOLO then outputs the predictions, which are drawn onto the original image and sent back to the client. The server runs using the same Docker image used for PoET, as the model needs all the libraries to run even in inference mode.

The client is a very simple script that opens a TCP connection to the server, waits for incoming images, and uses OpenCV[5] to display them.

5.4.2 PoET Inference Script

As we did for the backbone, we also developed a script for testing the transformer through a simple webcam (Figure 5.10). The client is again a Python client that, through OpenCV, accesses the camera, captures the images and sends them through a TCP socket to the server. The server takes the image and saves it to a pre-defined path. At each iteration, the new frame overwrites the oldest one. In this way, we can simply use a "while loop" that:

1. Loads the image from the path.
2. Feeds the image into the transformer.
3. Draws the prediction onto the image.
4. Sends the image to the client.
5. Updates the frame received from the client.

5.4.3 AR Visor Script

The two scripts described in Section 5.4.1 and Section 5.4.2 were mainly used for intermediate showcases, where our goal was to test the model's performance using a simple webcam. However, the final goal of the project is to create a framework capable of communicating with AR visors. In Section 5.4, we explained the decision to use the Varjo XR-4 over all other available visors. The idea was to convert the client script illustrated in Figure 5.10 into a C++ script capable of capturing Varjo images rather than simple webcam frames.

Once downloaded the official Varjo SDK and all the needed components, we compiled the example project provided by Varjo Inc. using Windows. By taking inspiration from a script available in the original SDK, we created the first script able to access the raw-cameras (Figure 5.11). A simple TCP-connection was then used to send each frame to the server.

[5]OpenCV is an open-source computer vision and machine learning software library.

```python
def client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 9999))

    cap = cv2.VideoCapture(0)

    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break

        # Send frame to server
        data = pickle.dumps(frame)
        message_size = struct.pack("Q", len(data))
        client_socket.sendall(message_size + data)

        # Receive processed frame from server
        data = b""
        payload_size = struct.calcsize("Q")

        while len(data) < payload_size:
            packet = client_socket.recv(4096)
            if not packet:
                break
            data += packet

        if len(data) < payload_size:
            break

        packed_msg_size = data[:payload_size]
        data = data[payload_size:]
        msg_size = struct.unpack("Q", packed_msg_size)[0]

        while len(data) < msg_size:
            data += client_socket.recv(4096)

        frame_data = data[:msg_size]
        frame = pickle.loads(frame_data)

        cv2.imshow('Processed Frame', frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    client_socket.close()
    cv2.destroyAllWindows()
```

Figure 5.10. Client's code used to access the webcam to retrieve real-time images and then display them once the server has sent back the image with the drawn predictions.

```cpp
// Get latest frame data
FrameData frameData{};
{
    std::lock_guard<std::mutex> streamLock(m_frameDataMutex);

    frameData.metadata = m_frameData.metadata;

    // Move frame datas
    for (size_t ch = 0; ch < m_frameData.colorFrames.size(); ch++) {
        frameData.colorFrames[ch] = std::move(m_frameData.colorFrames[ch]);
        m_frameData.colorFrames[ch] = std::nullopt;
    }

    frameData.cubemapFrame = std::move(m_frameData.cubemapFrame);
    m_frameData.cubemapFrame = std::nullopt;

    frameData.cubemapMetadata = m_frameData.cubemapMetadata;
}
```

(a)

Figure 5.11

```cpp
// Get latest color frame
for (size_t ch = 0; ch < frameData.colorFrames.size(); ch++) {
    if (frameData.colorFrames[ch].has_value()) {
        const auto& colorFrame = frameData.colorFrames[ch].value();

        // Skip conversions if the stream has only metadata.
        if (colorFrame.metadata.bufferMetadata.byteSize == 0) {
            continue;
        }

        // Convert datastream YUV frame to RGBA
        if (m_appState.options.undistortEnabled) {
            // We can downsample here to any resolution
            constexpr int c_downScaleFactor = 4;
            const auto w = colorFrame.metadata.bufferMetadata.width / c_downScaleFactor;
            const auto h = colorFrame.metadata.bufferMetadata.height / c_downScaleFactor;
            const auto rowStride = w * 4;

            std::optional<varjo_Matrix> projection = std::nullopt;

            // Convert to rectified RGBA in lower resolution
            std::vector<uint8_t> bufferRGBA(rowStride * h);
            DataStreamer::convertDistortedYUVToRectifiedRGBA(colorFrame.metadata.bufferMetadata,
                colorFrame.data.data(), glm::ivec2(w, h),
                bufferRGBA.data(), colorFrame.metadata.extrinsics,
                colorFrame.metadata.intrinsics, projection);

            // Send image to server
            sendImageToServer(sock, bufferRGBA, w, h);

            // Update frame data
            m_scene->updateColorFrame(static_cast<int>(ch), glm::ivec2(w, h),
                varjo_TextureFormat_R8G8B8A8_UNORM, rowStride, bufferRGBA.data());
        }
```

(b)

Figure 5.11

```cpp
SOCKET createAndConnectSocket(const std::string& serverIP, int serverPort)
{
    SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == INVALID_SOCKET) {
        std::cerr << "Socket creation failed: " << WSAGetLastError() << std::endl;
        WSACleanup();
        exit(EXIT_FAILURE);
    }

    sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(serverPort);
    inet_pton(AF_INET, serverIP.c_str(), &serverAddress.sin_addr);

    if (connect(sock, (sockaddr*)&serverAddress, sizeof(serverAddress)) == SOCKET_ERROR) {
        std::cerr << "Socket connection failed: " << WSAGetLastError() << std::endl;
        closesocket(sock);
        WSACleanup();
        exit(EXIT_FAILURE);
    }

    return sock;
}

void sendImageToServer(SOCKET sock, const std::vector<uint8_t>& bufferRGBA, int width, int height)
{
    cv::Mat img(height, width, CV_8UC4, const_cast<uint8_t*>(bufferRGBA.data()));
    std::vector<uchar> encodedImg;
    cv::imencode(".png", img, encodedImg);

    uint64_t dataSize = encodedImg.size();

    // Send the size of the encoded image first
    int bytesSent = send(sock, reinterpret_cast<const char*>(&dataSize), sizeof(dataSize), 0);
    std::cout << "Sent image size (" << sizeof(dataSize) << " bytes): " << bytesSent << " bytes sent." << std::endl;

    // Wait for ACK_SIZE from the server
    char ackSze[3] = { 0 };
    int bytesReceived = recv(sock, ackSze, 3, 0);
    if (bytesReceived <= 0 || std::string(ackSze, 3) != "SZE") {
        std::cerr << "Failed to receive size acknowledgment from server. " << std::string(ackSze, 3) << std::endl;
        return;
    }

    // Send the actual image data
    bytesSent = send(sock, reinterpret_cast<const char*>(encodedImg.data()), encodedImg.size(), 0);

    // Wait for ACK_IMAGE from the server
    char ackImg[3] = { 0 };
    bytesReceived = recv(sock, ackImg, 3, 0);
    if (bytesReceived <= 0 || std::string(ackImg, 3) != "IMG") {
        std::cerr << "Failed to receive image acknowledgment from server. " << std::string(ackImg, 3) << std::endl;
    }
}
```

(c)

Figure 5.11. The code shown in figure (a) is used to retrieve and update the current frame data. In figure (b) the frame data are then converted into an RGB image. Lastly, the code shown in figure (c) is used to instantiate a socket (called at system startup) and send the frame through the TCP conneciton.

Chapter 6

Test and Results

In this chapter, we will discuss the different configurations we used to perform the experiments. We will also present the obtained results, highlighting the pros and cons of each configuration.

6.1 Experiments' Configuration

The Table 6.1 and Table 6.2 represent each experiments' configuration.
The following terms will be used in the expermient tables:

- **Only Dummies**: The dataset consists only of objects whose pose needs to be regressed, leading to reduced occlusion in the scene.
- **Mixed**: The dataset consists of objects whose pose needs to be regressed and additional objects that are not part of the industrial object set. This introduces more occlusion in the dataset, thus training and testing the model in a more challenging environment.
- **Enhanced & Mixed**: A mixed dataset with further improvements to the synthetic data generation script to make the images appear more realistic.
- **Texturized-Enhanced & Mixed**: An "Enhanced & Mixed" dataset where objects are provided with textures instead of just a single gray paint.
- **NaN**: Indicates that the associated model was not trained. Namely, when NaN is appears in the YOLO column, then we trained PoET in "ground-truth mode". On the other hand, when we specify NaN in the PoET column, we simply trained YOLO without considering PoET.

Ground-truth training and bounding box training Training PoET using ground-truth rather than YOLO predictions has many pros and cons. The ground-truth label guarantees that there are no false detections; therefore, if an object is part of a frame, then its pose will be regressed. On the other hand, we cannot manage occluded objects. Indeed, if an object is inside the frame but completely occluded, the area described by its associated bounding box will represent the object which occlude and not the object associated with the label. This can obviously degrade the transformer's performance. Some datasets, such as YCB-Video used by PoET, have a parameter associated with each bounding box: *bbox_visible*. This represents the percentage of the non-occluded object (e.g., if $bbox_visible = 0.4$,

Test	Dataset Configuration	YOLO Hyperparameters	PoET Hyperparameters
I	Mixed	NaN	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 100
II	Mixed	NaN	Batch size: 16 Encoding layers: 4 Decoding layers: 4 Heads: 8 Epochs: 30
III	Only Dummies	NaN	Batch size: 8 Encoding layers: 4 Decoding layers: 4 Heads: 8 Epochs: 20
IV	Only Dummies (double size)	NaN	Batch size: 8 Encoding layers: 4 Decoding layers: 4 Heads: 8 Epochs: 20
V	Mixed	Batch size: 16 Learning rate: 0.01 Decay: 0.0005 3-Step Freezing	NaN
VI	Enhanced & Mixed	NaN	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 100
VII	Mixed	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 30%	NaN
VIII	Texturized-Enhanced & Mixed	NaN	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 100

Table 6.1. Configuration of Dataset, YOLO, and PoET Hyperparameters in different experiments.

Test	Dataset Configuration	YOLO Hyperparameters	PoET Hyperparameters
IX	Texturized-Enhanced & Mixed	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 15%	NaN
X	Texturized-Enhanced & Mixed	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 15%	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 170
XI	Realistic-Texturized-Enhanced & Mixed	NaN	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 170
XII	Realistic-Texturized-Enhanced & Mixed	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 15%	NaN
XIII	Realistic-Texturized-Enhanced & Mixed	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 15%	Batch size: 8 Encoding layers: 5 Decoding layers: 5 Heads: 16 Epochs: 130
XIV	Experiment XIII Fine-tuning	Batch size: 32 Learning rate: 0.01 Decay: 0.0005 Freezing: 15%	Batch size: 16 Encoding layers: 4 Decoding layers: 4 Heads: 16 Epochs: 150

Table 6.2. Configuration of Dataset, YOLO, and PoET Hyperparameters in different experiments.

then 60% of the pixels of the object are occluded). This value is used to eventually discard occluded objects within the frame. However, when dealing with synthetic datasets, retrieving such values can be very hard or even impossible. Therefore, we decided to set it to the percentage of the bounding box in the frame. Namely, if the pixels inside the bounding box area are 40% inside the frame and 60% outside the frame, then $bbox_visible = 0.4$. This obviously implies that we are assuming that if the object is in the frame, then it is not occluded. To balance this issue, we decided to train PoET also using the YOLO instance. This can partially solve the problem because if the object is occluded (or occluded within a fixed threshold), then the model will not detect it, and therefore the transformer will not learn any additional noise. Nevertheless, we still have to take into account the false positives that can be introduced by the YOLO model. To have a wider view of which model performs better, we will conduct the same experiments using PoET trained on ground-truth data and also with PoET trained using YOLO. Trivially, this implies that the experiments involving a YOLO instance will include an analysis dedicated to the detection results.

6.2 Experiments Analysis

In this section, we will discuss the results obtained with the experiment configurations listed in the previous section.

6.2.1 PoET Metrics

To expose the transformer results, we will use the following metrics:

- ADD-S (Average Distance of Model Points for Symmetrical Objects): It is a metric used to evaluate the accuracy of the estimated 6D pose of symmetrical objects. This metric accounts for the fact that, symmetrical objects can have multiple indistinguishable poses due to their symmetry. ADD-S of an object is defined as the average distance between each point on the 3D model using the ground truth pose and the closest point on the model using the estimated pose. Mathematically, it is given by:

$$\text{symmetricDistance}(obj, R, t, \hat{R}, \hat{t}) = \frac{1}{m} \sum_{x_1 \in M} \min_{x_2 \in M} \|(Rx_1 + t) - (\hat{R}x_2 + \hat{t})\| \tag{6.1}$$

  Where:

  - M is the set of 3D model points.
  - m is the number of points in the model.
  - R and t are the ground truth rotation and translation matrices.
  - $\hat{R}$ and $\hat{t}$ are the estimated rotation and translation matrices.
  - x_1 and x_2 are points in the 3D model.

  The pose is considered as correct if the symmetric distance is within a certain threshold, mathematically:

$$\text{correctPose}(obj, R, t, \hat{R}, \hat{t}) = \text{symmetricDistance}(obj, R, t, \hat{R}, \hat{t}) \leq \text{threshold} * d \tag{6.2}$$

Where d is the object diameter. In our case-study, the ADD-S of each object is reported as a real number in $[0, 100]$, given by:

$$\text{ADD-S}_{obj} = \frac{\sum_{(obj,R,t,\hat{R},\hat{t})\in Poses} \text{correctPose}(obj, R, t, \hat{R}, \hat{t})}{\text{number of poses}} * 100 \quad (6.3)$$

Trivially, in the tables presented in the following sections, the closer the ADD-S metric is to 100, the better the model's performance. An ADD-S of 100 (implicitly meaning 100%) indicates that all the poses are considered correct, as given by Equation 6.3.

- ADD (Average Distance of Model Points): It is a metric used to evaluate the accuracy of the estimated 6D pose of non-symmetrical objects. ADD is defined as the average distance between corresponding points on the 3D model using the ground truth and the estimated pose. Mathematically, it is given by:

$$\text{distance}(obj, R, t, \hat{R}, \hat{t}) = \frac{1}{m} \sum_{x\in M} \|(Rx + t) - (\hat{R}x + \hat{t})\| \quad (6.4)$$

Where:

  - M is the set of 3D model points.
  - m is the number of points in the model.
  - R and t are the ground truth rotation and translation matrices.
  - $\hat{R}$ and $\hat{t}$ are the estimated rotation and translation matrices.
  - x is a point in the 3D model.

The pose is considered as correct if the distance is within a certain threshold, mathematically:

$$\text{correctPose}(obj, R, t, \hat{R}, \hat{t}) = \text{distance}(obj, R, t, \hat{R}, \hat{t}) \leq \text{threshold} * d \quad (6.5)$$

Where d is the object diameter. In our case-study, the ADD of each object is reported as a real number in $[0, 100]$, given by:

$$\text{ADD}_{obj} = \frac{\sum_{(obj,R,t,\hat{R},\hat{t})\in Poses} \text{correctPose}(obj, R, t, \hat{R}, \hat{t})}{\text{number of poses}} * 100 \quad (6.6)$$

Like we stated for the ADD-S, having an ADD metric close to 100 (implicitly meaning 100%), indicates that all the poses are considered correct, as given by Equation 6.6.

- ARE (Average Rotation Error in degrees). Since it indicates an average, the optimal value is 0°. We must consider that this error is relative to all the axes. For example, if we achieve an ARE of 3°, it means that the average error in each rotation is 3°. However, in a 3D space, this rotation error could be distributed across the x-axis, y-axis, or z-axis. Typically, this error is equally distributed among the three axes.

- ATE (Average Translation Error in cm). Since it indicates an average, the optimal value is 0 cm. Exactly like ARE, the ATE is relative to all the axes, and we can assume that it is equally distributed among them.

In the following experiments, some of our objects are symmetrical, while some others are not. Therefore, for the symmetrical ones, we will use ADD-S as the metric, and for the non-symmetrical ones, we will use ADD. The ADD(-S) reported in the results tables indicates exactly this. We used three different thresholds to calculate the ADD(-S): 10%, 5%, and 2%. The ADD(-S) reported in the tables represents the weighted mean of these three thresholds, where the weight of each threshold is inversely proportional to the threshold value.

Note that all these metrics strongly depend on the distance from which the camera captures the frame. Specifically, having an ATE of 2 cm in a photo taken from 10 meters is different from having the same ATE in a photo taken from 1 meter. All the frames that compose the dataset have been generated using a distance $d \in [20 \text{ cm}, 60 \text{ cm}]$.

6.2.2 YOLO Metrics

Concerning the YOLO results, we will use the following metrics:

- Precision: Precision is the ratio of true positive detections to the total number of positive predictions. It measures the accuracy of the positive predictions made by the model. Mathematically, it is defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{6.7}$$

  The precision in this case is given as a real number in $[0, 1]$, where 0 indicates that the model has not detected any true positives, while 1 indicates that the model predicted only true positives and no false positives. In the tables, we expect to observe a precision as close to 1 as possible, but not exactly 1, as this could indicate that the model is overfitting.

- Recall: Recall is the ratio of true positive detections to the total number of actual positive instances. It measures the ability of the model to identify all relevant instances. Mathematically, it is defined as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{6.8}$$

  The recall in this case is given as a real number in $[0, 1]$, where 0 indicates that the model has not detected any true positives, while 1 indicates that the model predicted only true positives and no false negatives. In the tables, we expect to observe a recall as close to 1 as possible, but not exactly 1, as this could indicate that the model is overfitting.

- mAP@50: Mean Average Precision at IoU threshold 0.5 (mAP@50) is the mean of the average precision values for all classes, where a prediction is considered correct if the Intersection over Union (IoU) with the ground truth is at least 0.5. It provides a single metric to evaluate the performance of the object detector. The value is represented as a real number in $[0, 1]$, where 0 means that the predicted bounding boxes do not overlap at all with the ground-truth bounding boxes, and 1 means that they exactly coincide. In the tables, we expect to observe an mAP@50 as close to 1 as possible, but not exactly 1, as this could indicate that the model is overfitting.

- mAP@95: Mean Average Precision at IoU thresholds from 0.5 to 0.95 (mAP@95) is the mean of the average precision values for all classes, averaged over multiple IoU thresholds ranging from 0.5 to 0.95 in increments of 0.05. This metric provides a more comprehensive evaluation of the detector's performance across different levels of localization precision. Mathematically, it is defined as:

$$\text{mAP@95} = \frac{1}{N}\sum_{i=1}^{N}\text{AP}_i \tag{6.9}$$

  where N is the number of IoU thresholds (10 in this case: 0.5, 0.55, 0.6, ..., 0.95), and AP_i is the average precision at the i-th IoU threshold.

  Example: Suppose we have an object detection model evaluated on a dataset with IoU thresholds ranging from 0.5 to 0.95. For each IoU threshold, we calculate the AP for each class. For instance, if we have two classes and the following AP values at each threshold:

IoU Threshold	AP for Class 1	AP for Class 2
0.5	0.85	0.80
0.55	0.83	0.78
...	...	...
0.95	0.60	0.55

Table 6.3. Example AP values at different IoU thresholds for two classes.

  The mAP for each class would be the mean of the AP values across all thresholds. The final mAP@95 is the mean of these class-wise mAP values. The value is represented as a real number in $[0, 1]$, where 0 means that the predicted bounding boxes do not overlap at all with the ground-truth bounding boxes, and 1 means that they exactly coincide. In the tables, we expect to observe an mAP@95 as close to 1 as possible, but not exactly 1, as this could indicate that the model is overfitting.

- Objectness Loss: Objectness Loss measures the error in predicting whether a bounding box contains an object or not. It is a component of the YOLO loss function that ensures the model correctly identifies boxes containing objects. We expect Objectness Loss to decrease over time as the model improves its ability to distinguish between boxes containing objects and those that do not.

- GIoU Loss: Generalized Intersection over Union (GIoU) Loss measures the difference between the predicted and ground truth bounding boxes. It is an improved version of IoU loss that also takes into account the distance between the predicted and ground truth boxes, providing a better gradient for training. We expect GIoU Loss to decrease as the model becomes better at aligning the predicted bounding boxes with the ground truth, reflecting more accurate bounding box predictions.

- Classification Loss: Classification Loss measures the error in predicting the correct class label for each bounding box. It is a component of the YOLO loss function that ensures the model correctly classifies the detected objects. We expect Classification Loss to decrease as the model learns to accurately classify the objects, with lower values indicating better performance.

6.2.3 Experiment I and II (PoET Training)

These two experiments are grouped because both their configurations and results are very similar, so it makes sens to discuss them together. These configurations are the most basic ones, with a mixed dataset, thus containing both dummies and other objects used to make some noise, and ground-truth data. Since the mixed dataset is very similar to the one used in the original paper [47], in terms of number of instances, we decided to use the same hyperparameters. In the second experiment we slightly changed the batch size to check if this could enhance or not the transformer performance. However, both of these two experiments achieved very bad results on test set (Table 6.4). After further analysis, we found out that the model was highly overfitting on training set. To determine that the transformer was overfitting, we set both training and testing set with the same dataset and the results were nearly perfect. So these configurations were discarded due to this problem.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	6.27	5.40	8.20	12.50	132.32	5.75
DUMMY#1-2	2.89	2.10	3.20	5.00	127.88	10.73
DUMMY#5-2	21.79	16.00	22.50	32.50	131.09	7.97
DUMMY#5-3	47.57	38.00	50.00	65.00	123.69	4.96
DUMMY#6-1	0.94	0.80	1.20	2.00	97.10	46.24
DUMMY#6-2	14.76	11.00	15.00	22.00	119.14	6.15
DUMMY#1-3	4.27	3.50	5.00	7.50	125.99	10.08
DUMMY#2-1	2.22	1.80	2.50	4.00	120.11	15.45
DUMMY#2-2	2.59	2.00	3.00	4.50	121.25	12.16
DUMMY#3-1	2.67	2.20	3.10	4.60	119.78	11.87
DUMMY#3-2	4.06	3.50	4.80	6.00	121.46	6.81
DUMMY#4-1	8.53	7.00	9.00	12.00	129.83	9.14
DUMMY#4-2	17.00	14.00	18.00	24.00	124.23	12.42
DUMMY#5-1	3.78	3.00	4.00	6.00	122.24	10.34
Average	**9.95**	**7.42**	**10.15**	**14.32**	**123.86**	**10.01**

Table 6.4. This table shows the ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment I on the testing set. The results for the experiment II are very similar so they will not be reported.

6.2.4 Experiment III (PoET Training)

To address the problems encountered in the first two experiments, the new configuration was set such that the transformer would be facilitated during the training. In particular, we generated a new dataset in which each sequence was composed of only two dummies. The idea was to avoid as much as possible the problem derived from the object occlusion described in Section 6.1. Nevertheless, even using this simplified dataset and by reducing the number of epochs, we noticed that the model was still overfitting. In particular, to check the behavior during the training, we run the evaluation step after each training epoch. After only 5 epochs (during which the model was still not overfitting but also unable to predict something with a reasonable precision), the transformer started to overfit again.

6.2.5 Experiment IV (PoET Training)

The fourth experiment was mainly focused on the dataset size. One of the problems that can lead the model to overfit is not only the quality of the data, but also the number of instances used during the training. Therefore, we doubled the size of the dataset containing only dummies. Again, the achieved results were almost identical to the previous experiment.

6.2.6 Experiment V (YOLO Training)

Since all the previous 4 experiments could be considered failures, we tried a totally different approach. First, we decided to use the mixed dataset (like what we did in the first two experiments); nevertheless, we did not train PoET in this experiment but focused on obtaining a well-trained YOLO model. Our idea was to use this checkpoint to train the PoET transformer in "backbone mode" rather than "ground-truth mode" as we did during the first 4 experiments. This experiment can be split into three steps:

1. By sending an email to the PoET paper's author, we obtained the original YOLO pre-trained model. In particular, that model was trained on ImageNet and YCB-Video datasets. Therefore, we froze the first 70% of the layers (Section 5.3.1) and trained YOLO only on 30 sequences. The goal was to perform a quick training, aiming to learn the high-level pattern of objects in the custom dataset (the higher the number of frozen layers, the fewer the layers to be trained, and therefore, the less time needed to complete an epoch). The training ran for 57 epochs, considering that the model training resumed from epoch 13, as the checkpoint we were provided with had stopped at the 12th epoch. We can notice in Figure 6.1 that each graph presents a peak at epoch 13 due to the fact that from epoch 0 to epoch 12 the stored values were those achieved during the previous training (the one which produced the checkpoint we are starting with) and since it was not completed, these values were still part of the graph. Moreover, as we can see from the graphs, all the values were already converged around the 50th epoch, except the precision, which was still not completely stalling.

2. After the first preliminary training, the goal was to repeat the process by training the previously obtained checkpoint until reaching a final step checkpoint in which no layers were frozen. This was done to achieve step-by-step a more finely tuned model. Obviously, to train more layers, the model needs more instances to fine-tune not only the last layers (the 30% left non-frozen in the previous step) but also the newly unfrozen ones. So we increased the dataset from 30 up to 80 sequences and froze 50% of the layers. The model was trained for 20 epochs. Unlike Figure 6.1, in Figure 6.2 we can see that there are no peaks in the graphs since the previous training was considered complete, hence no evaluation values were stored in the model weights. In this specific case, we can see that not all the metrics were stalling at the 20th epoch, therefore the model could be run for more epochs. However, this was not a problem as we trained the model for an additional step.

3. The third step consisted of increasing the number of sequences from 80 up to 220 and freezing only 30% of the layers. We ran the training for 40 epochs. As we can see in Figure 6.3, the values do not seem to be stalling. However, by taking a close look at the scale on the y-axis of each graph, we can see that

the values have improved but within a very small range. Therefore, 40 epochs were sufficient to perform this fine-grained tuning.

At the end of these three steps, we achieved a well-trained YOLO model, as we can see in Table 6.5. During each step, we were able to improve at least one of the metrics used to track model performance, resulting in 88.8% precision, 95.39% recall, 96.39% mAP@50, and 78.73% mAP@95. Although training in multiple steps may not have led to significantly better results, it allowed us to perform step-by-step training, where we could adjust the parameters between each step. This approach led to three micro-trainings rather than one large training session. If we had chosen incorrect parameters for the large training, we would have had to retrain everything from scratch. Whereas with micro-training, we could repeat only the affected step, ultimately saving time.

Training Step	**Precision**	**Recall**	**mAP@50**	**mAP@95**
1	0.7972	0.9268	0.9352	0.7343
2	0.8854	0.9319	0.9448	0.7573
3	0.8880	0.9539	0.9639	0.7873

Table 6.5. Performance metrics for YOLO model: Precision, Recall, mAP@50, and mAP@95. Note that all the metrics are slowly tending to 1, this indicates that the steps were useful to improve all the metrics. The precision is the metric mainly improved during these steps, thus indicating both the increment of true positives and decrement of false positives.

Training Step	**Classification Loss**	**Objectness Loss**	**GIoU Loss**
1	0.01964	0.01716	0.005578
2	0.01765	0.01732	0.003069
3	0.01580	0.01534	0.001548

Table 6.6. Performance metrics for YOLO model: Classification Loss, Objectness Loss, and GIoU Loss. Although we observe only minor improvements in the Classification and Objectness Losses, there is a significant reduction in the GIoU Loss. This indicates that the predicted bounding boxes are fitting the objects in the frame more accurately.

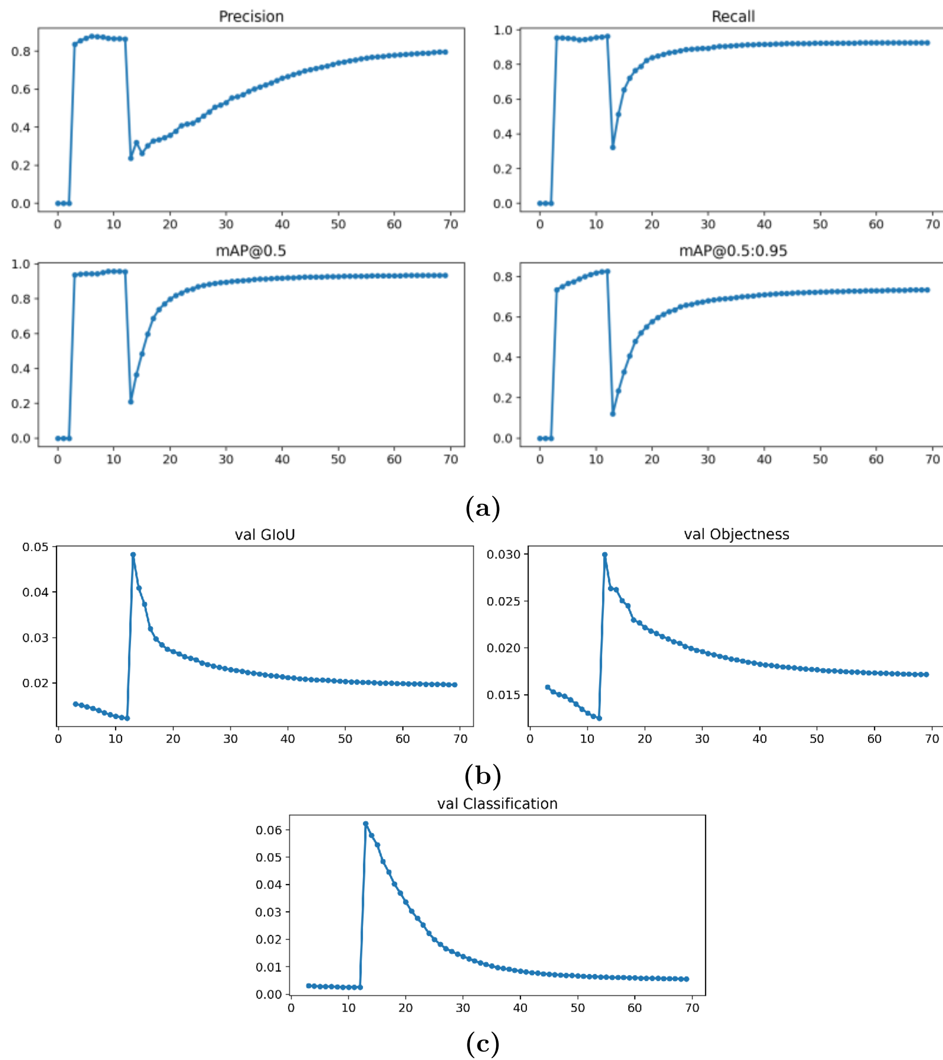


Figure 6.1. All these graphs are relative to the first step of the YOLO training during the experiment V. Graphs representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). The peaks that can be observed in each of the three figures around epoch 13 are caused by the transfer learning technique used during this step. From epoch 0 to epoch 12 the figures are representing the values stored in the checkpoint, therefore from epoch 13 the values are relative to our dataset. Note that, in figure (a) the more the lines are close to 1, the better the model is. In figure (b) and (c), as we are treating losses, the lines tend to 0. Obviously, this trend is not perfect since is the first step of the experiment.

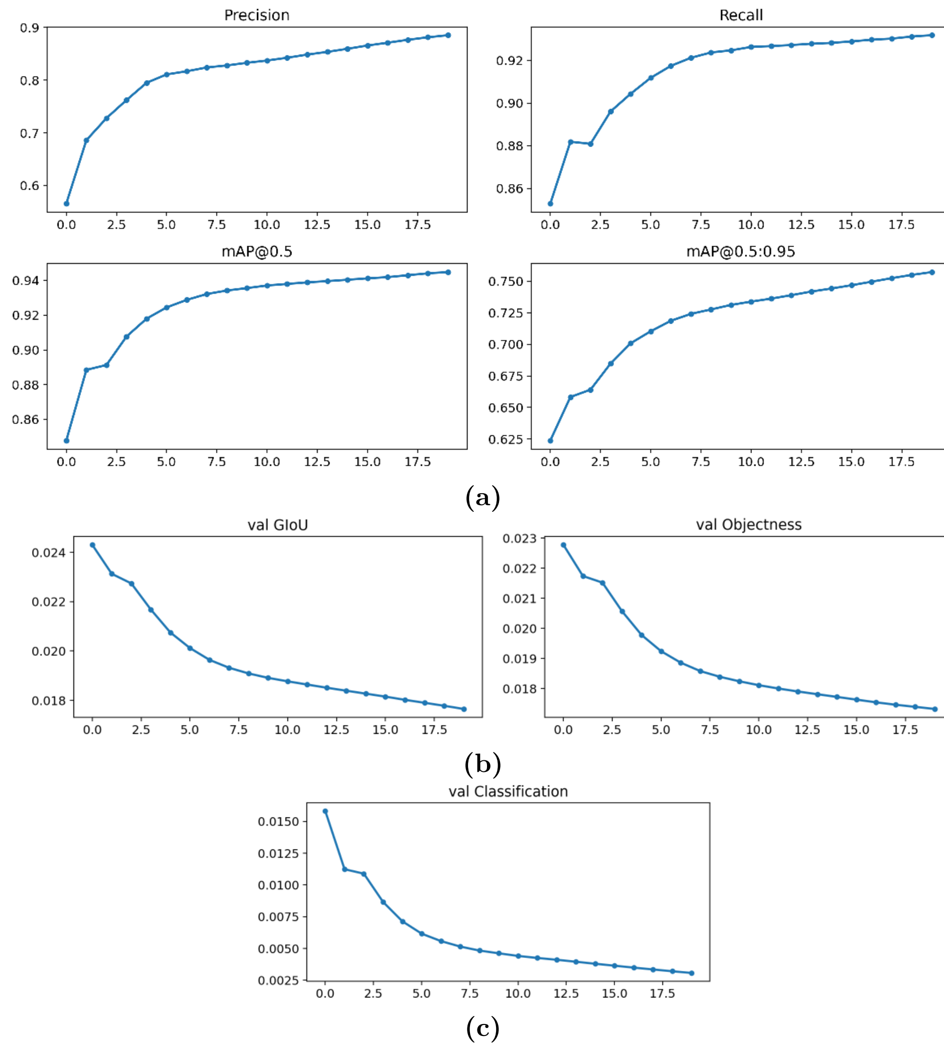


Figure 6.2. All these graphs are relative to the second step of the YOLO training during the experiment V. Graphs representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). Note that the steps on the y-axe are very small, thus indicating that the model is already close to its best performance.

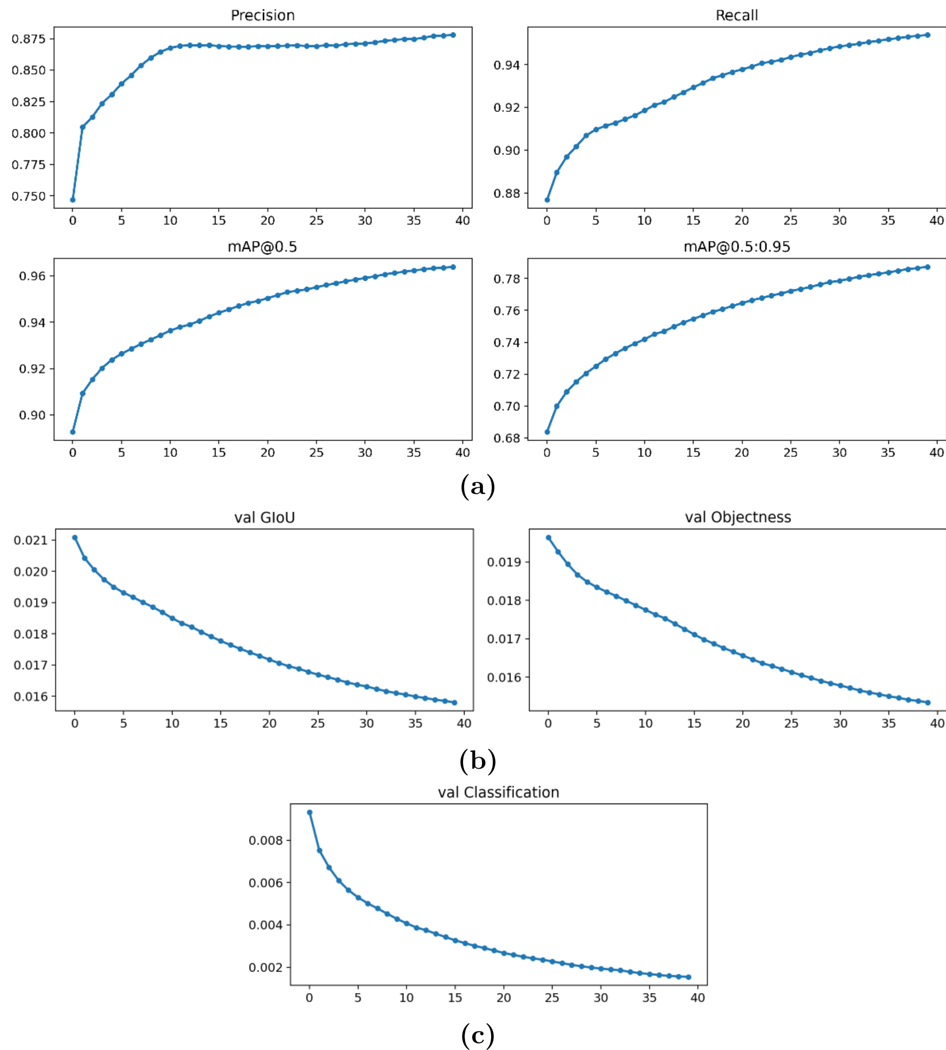


Figure 6.3. All these graphs are relative to the third step of the YOLO training during the experiment V. Graphs representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). Note that the steps on the y-axe are very small, thus indicating that the model is already close to its best performance.

6.2.7 Experiment VI (PoET Training)

For this experiment, we decided to delve deeper into the problems encountered during transformer training. One of our hypotheses, was that the script used to convert the dataset from the original YCB-Video format into the BOP one, was introducing some kind of error. After reviewing the code, we fixed it and also enhanced the generated images to make them look more realistic. This experiment involved generating a new dataset with 600 scenes and training PoET again using the hyperparameters from Experiment I. Finally, we achieved almost acceptable results, as shown in Table 6.7. However, they were far from the results shown in the original paper [47]. As shown in Figure 6.4, the loss, once it reached epoch 80, was no longer converging at a reasonable speed. We also tried to train the model up to epoch 120; however, the results did not improve.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	71.67	31.31	92.08	99.26	73.26	3.67
DUMMY#1-2	66.40	29.19	82.39	98.79	60.50	8.54
DUMMY#5-2	74.84	52.28	90.67	98.67	88.02	6.48
DUMMY#5-3	74.92	47.37	93.81	99.40	125.91	3.95
DUMMY#6-1	29.25	1.24	28.18	67.29	50.40	43.48
DUMMY#6-2	78.91	59.97	96.73	99.61	58.75	4.45
DUMMY#1-3	54.89	14.62	62.79	96.87	42.43	6.21
DUMMY#2-1	62.50	23.14	77.97	97.03	68.12	12.37
DUMMY#2-2	58.11	16.14	66.48	97.91	91.83	10.89
DUMMY#3-1	69.53	38.83	85.37	97.53	32.71	12.37
DUMMY#3-2	61.40	24.85	73.88	95.59	77.95	11.49
DUMMY#4-1	75.90	49.90	94.59	99.33	46.30	5.97
DUMMY#4-2	77.51	61.13	92.49	98.97	39.93	4.37
DUMMY#5-1	66.86	27.00	84.38	98.98	60.44	8.26
Average	**65.91**	**34.07**	**80.13**	**96.09**	**66.27**	**7.77**

Table 6.7. This table shows the ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment VI on the testing set, along with accuracy at different thresholds.

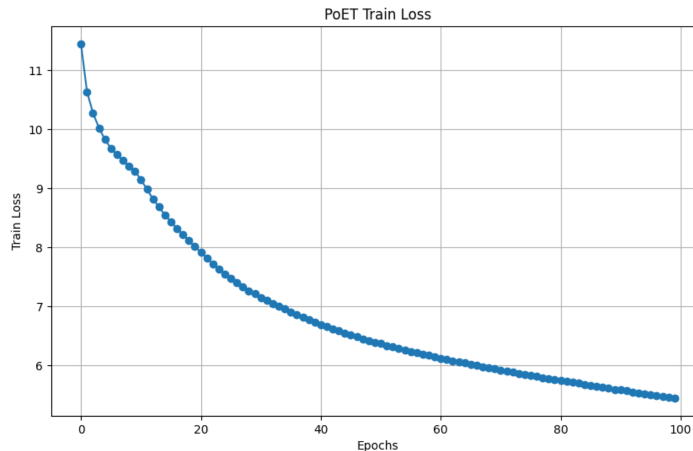


Figure 6.4. Graph showing the loss trend during the training in experiment VI. The trend is generally downward. Moreover, since the line is not clearly leveling off, this suggests that the model could be improved by increasing the number of epochs.

6.2.8 Experiment VII (YOLO Training)

During this experiment, we trained YOLO starting from the checkpoint obtained in Experiment 5. As the main strategy, we froze 30% of the layers, running the process for 100 epochs. During this experiment, we also added the capability to see the results per class.

We trained the object detector again as we modified the synthetic generator during Experiment 6. Therefore, the appearance of the images changed significantly, necessitating retraining. By comparing the results achieved in Table 6.8 with those described in Table 6.5, the overall precision of the model with the semi-realistic images decreased by 0.08, while the other metrics improved. The greatest improvement was in the mAP@95, which increased from 0.7873 to 0.9197.

As we can note from the graphs shown in Figure 6.5, the model has correctly learned until it reached epoch 80, then it started to show some overfitting symptoms. However, this is not a problem as we selected the 80th epoch as the best, so we saved that epoch's checkpoint.

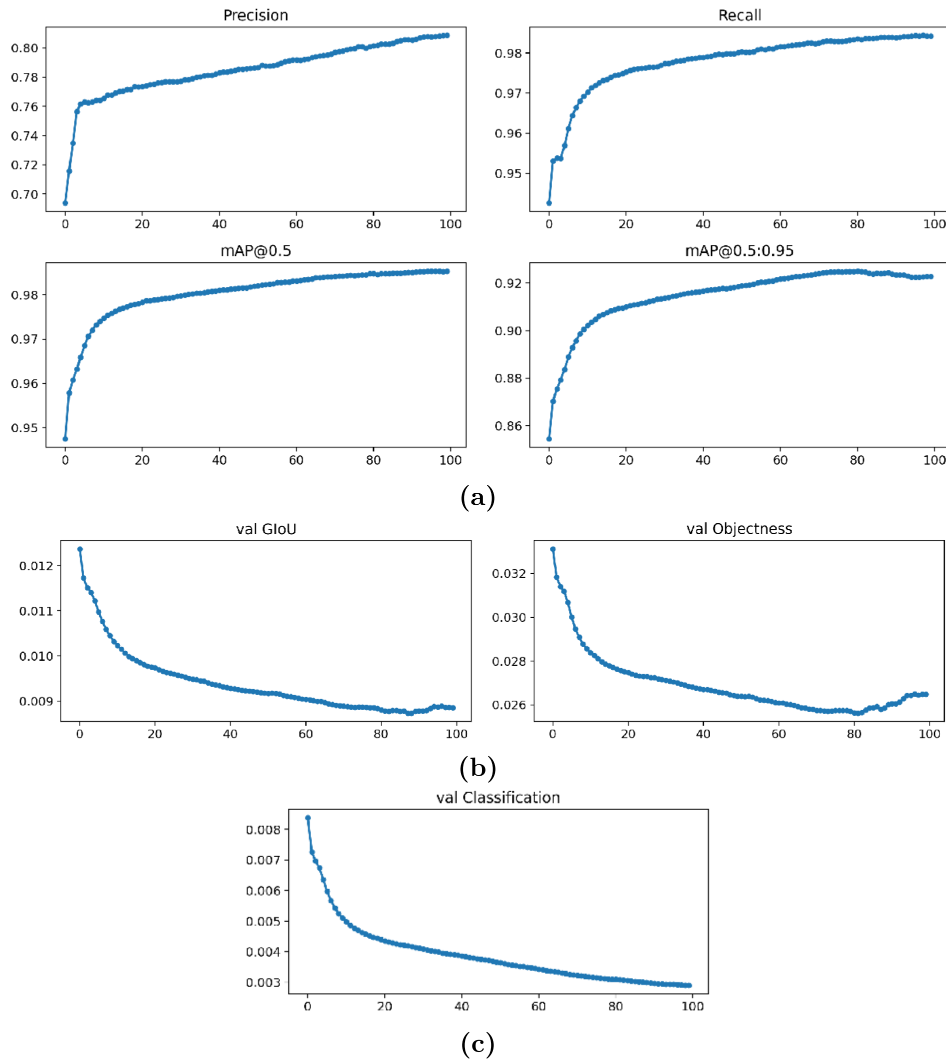


Figure 6.5. Graphs of the experiment VII representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). These figures suggest that around epoch 80, the model began to overfit. This is evident from Figure (b), where the two test losses started to increase again around epoch 80, rather than continuing to decrease. Combined with the similar behavior observed in Figure (a), this indicates that the best model is likely the one obtained at epoch 80.

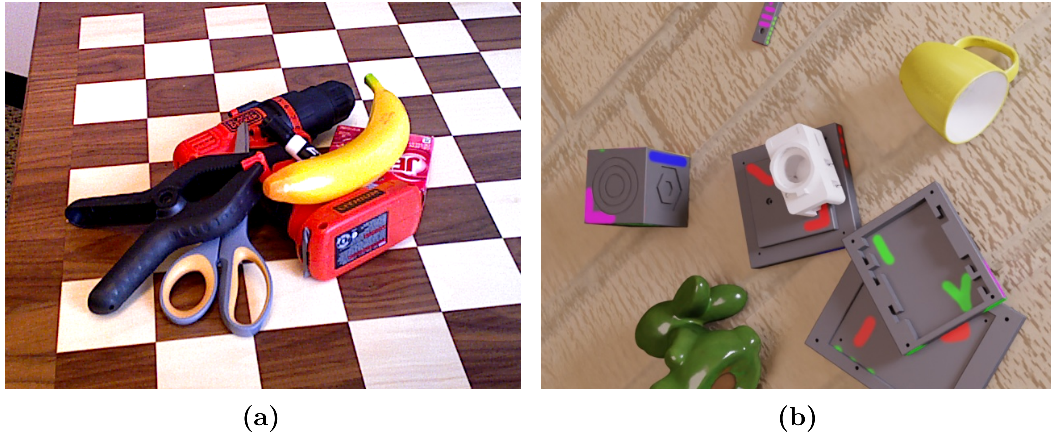
Object Name	**Precision**	**Recall**	**mAP@0.5**	**mAP@0.5:0.95**
DUMMY#1-1	0.8027	0.9907	0.9875	0.9506
DUMMY#1-2	0.7498	0.9877	0.9841	0.9439
DUMMY#5-2	0.7925	0.9690	0.9795	0.8746
DUMMY#5-3	0.8494	0.9917	0.9928	0.9170
DUMMY#6-1	0.8286	0.9758	0.9792	0.9174
DUMMY#6-2	0.9045	0.9990	0.9942	0.9775
DUMMY#1-3	0.7679	0.9804	0.9838	0.9245
DUMMY#2-1	0.7647	0.9808	0.9822	0.9267
DUMMY#2-2	0.8446	0.9917	0.9903	0.9275
DUMMY#3-1	0.8133	0.9843	0.9863	0.9148
DUMMY#3-2	0.7656	0.9597	0.9661	0.8355
DUMMY#4-1	0.8575	0.9931	0.9937	0.9529
DUMMY#4-2	0.8216	0.9861	0.9907	0.9092
DUMMY#5-1	0.7582	0.9882	0.9839	0.9453
Average	**0.8038**	**0.9857**	**0.9861**	**0.9197**

Table 6.8. YOLO performance metrics for each class and their average, achieved in experiment VII.

6.2.9 Experiment VIII (PoET Training)

This experiment was mainly focused on understanding the main differences between our dummies and the objects used for the original PoET implementation. As we can see from the comparison in Figure 6.6, the original objects were all provided with textures while ours were all colored using solid gray[1]. In the industrial environment, having all the 3D-printed objects with the same tint can save both time and money as they don't need to be painted in specific ways. However, it is reasonable to think that having no texture on the dummies can make it almost impossible in some cases to understand the orientation of the object. Therefore, for this experiment, we manually drew some markers as shown in Figure 6.6. The goal of drawing these markers such that they can be easily distinguished was to provide an "upper bound" to the company. With Experiment 6, we provided a "lower bound"; therefore, using more realistic and industrial markers (such as an "R" to indicate right, "L" to indicate left, "B" for the bottom part, etc.), the achieved results must lie between these two limits obtained in Experiment 6 and this one. Then we trained again the model on the dataset composed with these new objects, the results are shown in Table 6.9.

[1]Solid gray is the standard gray graduation used in Blender and most 3D modeling software.



(a) (b)

Figure 6.6. In figure (a) we can see a frame taken from the original YCB-Video dataset, while in figure (b) we can observe a synthetic frame with the solid-colored dummies.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	73.00	42.40	90.35	98.80	23.20	3.68
DUMMY#1-2	68.90	43.06	81.36	97.54	25.25	5.26
DUMMY#5-2	72.24	51.67	85.28	96.82	33.26	4.86
DUMMY#5-3	80.00	65.90	95.48	99.24	30.12	3.05
DUMMY#6-1	35.50	11.03	40.27	64.19	24.33	3.26
DUMMY#6-2	82.14	72.89	96.62	99.55	21.24	3.43
DUMMY#1-3	60.89	30.86	71.06	95.37	26.33	5.08
DUMMY#2-1	63.98	36.57	76.15	93.83	23.65	6.12
DUMMY#2-2	68.22	42.24	79.92	97.08	23.02	5.19
DUMMY#3-1	66.44	37.68	79.66	96.08	24.35	6.17
DUMMY#3-2	56.76	26.55	66.44	90.32	36.26	7.31
DUMMY#4-1	76.44	54.53	93.56	99.05	22.53	4.47
DUMMY#4-2	77.28	62.70	91.18	98.41	27.58	4.16
DUMMY#5-1	71.39	46.04	85.55	98.11	21.97	4.92
Average	**68.09**	**44.58**	**80.92**	**94.60**	**25.78**	**4.78**

Table 6.9. This table shows the updated ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment on the testing set, along with the accuracy at different thresholds in the experiment VIII.

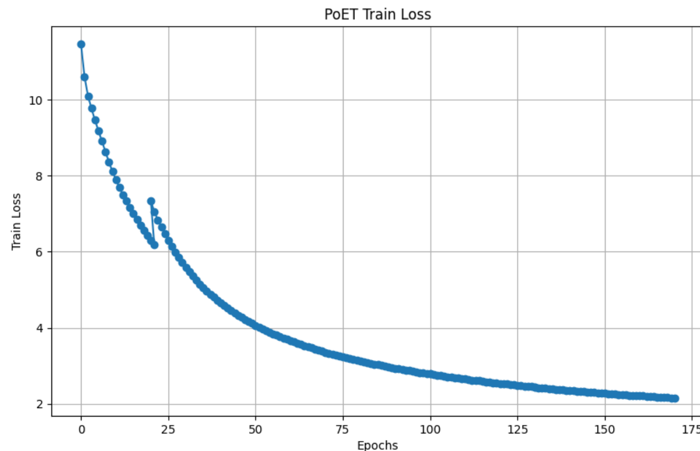


Figure 6.7. Graph showing the loss trend during the training in experiment VIII. The peak observed around epoch 22 occurred because the training was paused and resumed on another machine due to technical issues.

6.2.10 Experiment IX (YOLO Training)

As already seen in previous experiments, once we created a new dataset for training the transformer in Experiment 8, we also trained YOLO on the same dataset, using the model achieved in Experiment 7 as the starting point for the training by also freezing 15% of the layers. The training lasted for 100 epochs, and the results are visible in Table 6.10. As we can see from the table, we improved the average precision by a factor of 10%, while keeping the other metrics close to perfection. The markers added to the object in the previous experiment helped YOLO achieve better detection. This was trivially predictable by observing how much PoET benefited from the textures.

Object Name	Precision	Recall	mAP@0.5	mAP@0.5:0.95
DUMMY#1-1	0.8994	0.9978	0.9919	0.9732
DUMMY#1-2	0.8992	0.9982	0.9933	0.9715
DUMMY#5-2	0.8831	0.9925	0.9935	0.9187
DUMMY#5-3	0.9014	0.9972	0.9947	0.9303
DUMMY#6-1	0.9023	0.9988	0.9933	0.9768
DUMMY#6-2	0.9383	0.9994	0.9945	0.9888
DUMMY#1-3	0.8668	0.9943	0.9924	0.9637
DUMMY#2-1	0.8830	0.9976	0.9930	0.9585
DUMMY#2-2	0.9031	0.9968	0.9937	0.9564
DUMMY#3-1	0.8748	0.9940	0.9915	0.9460
DUMMY#3-2	0.8769	0.9838	0.9891	0.8855
DUMMY#4-1	0.9190	0.9985	0.9943	0.9744
DUMMY#4-2	0.8889	0.9974	0.9946	0.9381
DUMMY#5-1	0.8988	0.9973	0.9928	0.9741
Average	**0.8993**	**0.9965**	**0.9933**	**0.9565**

Table 6.10. YOLO performance metrics for each class and their average, achieved in experiment IX.

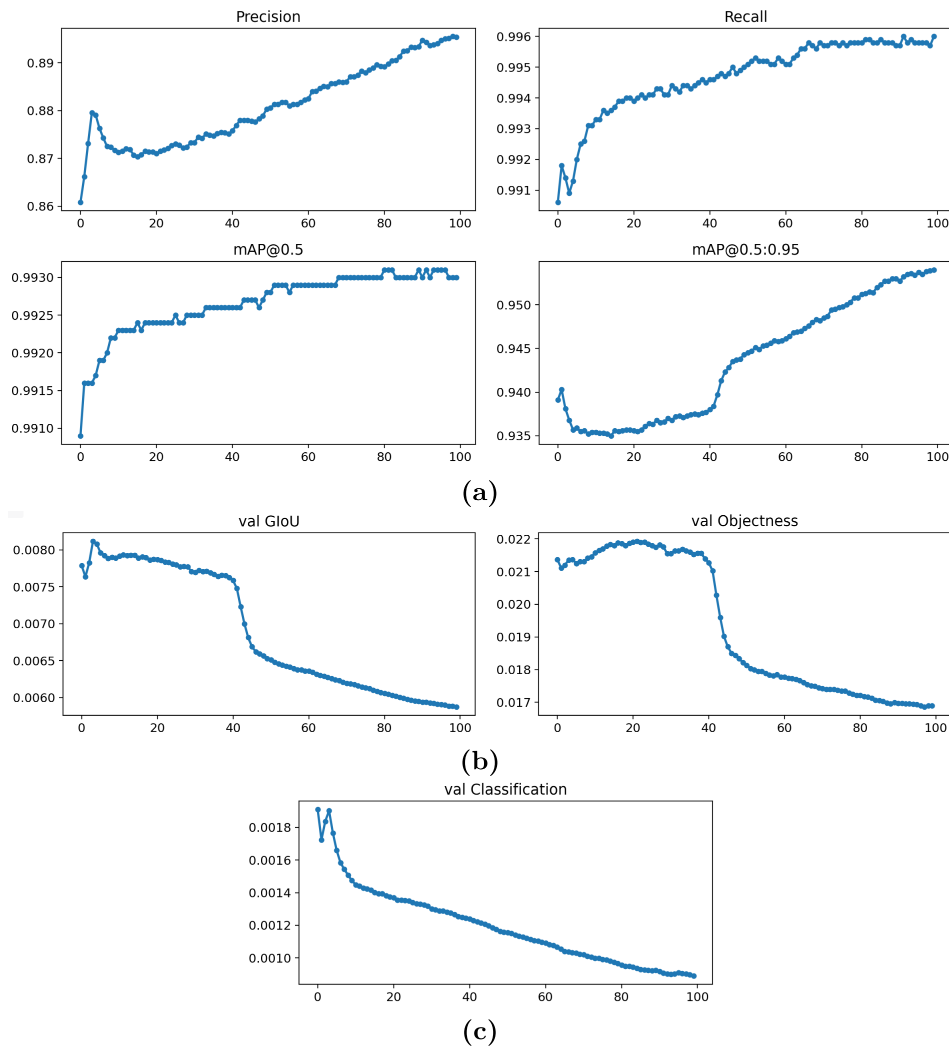


Figure 6.8. Graphs of the experiment IX representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). The trends observable in these figures are accurate: the losses tend to approach 0, while the other metrics approach 1. Although the plots may appear unusual due to the lack of straightforward improvement, a closer examination of the y-axis reveals the small magnitude of the changes. This suggests that the model has reached its peak performance, and therefore, even small fluctuations can result in significant changes in the graph.

6.2.11 Experiment X (PoET Training with YOLO)

Once we obtained a satisfactory version of both YOLO (Experiment 9) and PoET (Experiment 8) ground-truth models, we decided to retrain PoET in backbone mode. In this experiment, the labels used for training the model were not the actual ground-truth labels but rather those predicted by the YOLO backbone. The backbone weights used were from Experiment 9.

Intuitively, the results should be worse compared to the ground-truth model due to the additional backbone inference step, which may fail to predict objects accurately or output incorrect classification predictions. The results are presented in Table 6.11. The overall ARE is approximately 10 degrees worse than the ground-truth results. We also expected this trend to be followed by the ADD(-S) metric; however, interestingly, the distance metrics improved despite the degradation in the rotation metrics.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	78.06	51.86	96.26	99.78	28.38	2.84
DUMMY#1-2	75.76	54.29	90.74	99.52	25.67	4.70
DUMMY#5-2	79.74	65.05	93.57	99.15	38.13	4.20
DUMMY#5-3	78.73	63.67	95.57	99.37	113.12	3.44
DUMMY#6-1	31.26	9.98	35.13	57.26	35.13	2.66
DUMMY#6-2	84.40	79.40	98.35	99.88	28.30	3.12
DUMMY#1-3	69.27	42.08	81.43	98.89	21.61	4.14
DUMMY#2-1	73.29	51.25	87.89	98.06	26.54	6.14
DUMMY#2-2	76.18	56.81	88.92	99.50	29.06	4.96
DUMMY#3-1	76.75	57.51	91.45	98.83	19.44	4.92
DUMMY#3-2	66.46	37.10	78.88	95.93	47.11	7.31
DUMMY#4-1	80.52	67.00	96.41	99.66	27.32	4.25
DUMMY#4-2	80.67	70.22	94.58	99.27	29.83	3.67
DUMMY#5-1	76.92	56.48	92.38	99.44	25.89	4.55
Average	**73.43**	**54.48**	**87.25**	**96.04**	**35.43**	**4.30**

Table 6.11. This table shows the updated ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment on the testing set, along with the accuracy at different thresholds in the experiment X.

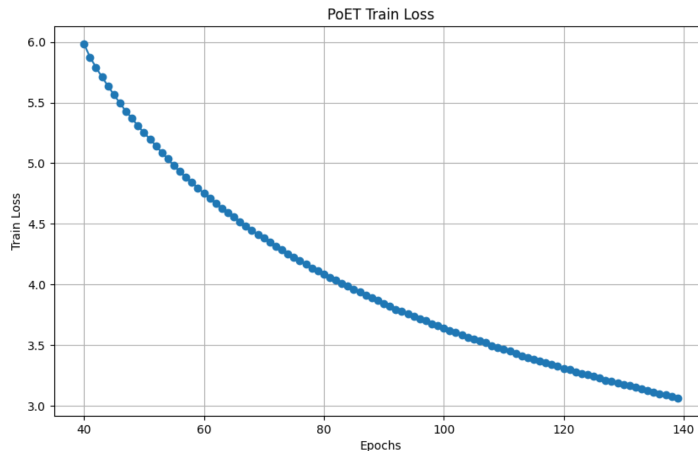


Figure 6.9. Graph showing the loss trend during the training in experiment X. Although the line does not seem to be converging, the model has not been learning effectively. The improvements in terms of loss are too small relative to the number of epochs required to achieve them.

6.2.12 Experiment XI (PoET Training)

The results shown in Experiment 6 and Experiment 9 can be used as worst-best case scenarios. However, in the industries the textures used for the best-case scenario are not reproducible. The last version of our dataset is the one which tries to replicate realistic textures in which each object's face is marked with a red symbol indicating the 3D axe direction towards the face is pointing to. Namely, the symbols can be:

- +X
- -X
- +Y
- -Y
- +Z
- -Z

You can see some examples of the new textures in Figure 6.11.

From the results shown in Table 6.12, we expected results to lie between those presented in Experiment 8 and those shown in Experiment 6. However, the average ARE is about 3 degrees lower than the "optimal" results we achieved in Experiment 8. This means that more complex textures do not necessarily yield better results. By using these simple yet efficient textures, we achieved the best results and provided a ready-for-industrial-use model.

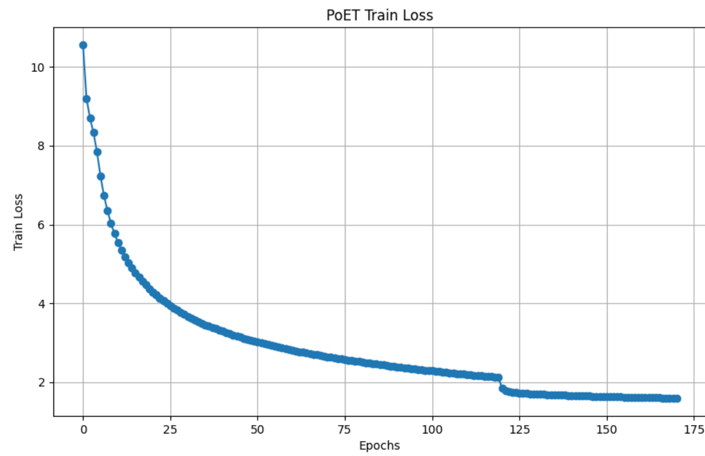


Figure 6.10. Graph showing the loss trend during the training in experiment XI. The jump observable around epoch 120 occurred because the training was paused and resumed on another machine due to technical issues.

(a) **(b)**

Figure 6.11. In the two figures we can see the examples of industrial textures used in experiment XI. Each object is clearly marked according to industrial convention. This provides a more realistic example, as these markers can be easily replicated during the piece creation process. The solid color used for the object is the default blender color, however, it may be adjusted during the production.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	82.10	65.75	97.54	99.84	17.55	2.42
DUMMY#1-2	80.17	67.01	93.52	99.66	18.59	3.79
DUMMY#5-2	80.37	68.39	92.78	98.82	22.28	3.54
DUMMY#5-3	83.17	73.65	96.88	99.67	34.68	2.62
DUMMY#6-1	48.35	25.00	57.42	73.63	22.39	2.19
DUMMY#6-2	86.03	83.15	98.41	99.87	16.90	2.66
DUMMY#1-3	73.38	52.25	86.13	98.60	15.31	3.43
DUMMY#2-1	77.73	61.06	92.16	98.94	19.74	4.89
DUMMY#2-2	77.63	61.72	88.84	99.46	33.96	5.54
DUMMY#3-1	78.52	62.46	92.07	98.78	27.80	5.84
DUMMY#3-2	68.59	42.83	80.59	96.47	33.79	6.36
DUMMY#4-1	82.65	74.30	96.89	99.55	20.66	3.61
DUMMY#4-2	82.25	74.54	95.31	99.43	23.05	3.32
DUMMY#5-1	81.09	67.70	94.80	99.73	14.37	3.54
Average	**77.29**	**62.84**	**90.24**	**97.32**	**22.70**	**3.80**

Table 6.12. This table shows the updated ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment on the testing set, along with the accuracy at different thresholds in the experiment XI.

6.2.13 Experiment XII (YOLO Training)

As we did with all the other datasets, we also trained the YOLO model on the realistic-texture dataset. The checkpoint used was the one achieved in Experiment 7 with a freezing value of 15%. The results are shown in Table 6.13. Similar to Experiment 11, where we improved the results obtained with the non-realistic textures, we observed performance gains in each model during this training as well.

Object Name	Precision	Recall	mAP@0.5	mAP@0.5:0.95
DUMMY#1-1	0.9106	0.9974	0.9930	0.9764
DUMMY#1-2	0.8866	0.9960	0.9931	0.9739
DUMMY#5-2	0.8977	0.9912	0.9937	0.8935
DUMMY#5-3	0.9009	0.9981	0.9948	0.9000
DUMMY#6-1	0.9193	0.9984	0.9936	0.9780
DUMMY#6-2	0.9437	0.9996	0.9945	0.9866
DUMMY#1-3	0.8910	0.9945	0.9933	0.9673
DUMMY#2-1	0.8969	0.9955	0.9931	0.9594
DUMMY#2-2	0.9134	0.9974	0.9933	0.9585
DUMMY#3-1	0.8885	0.9925	0.9918	0.9523
DUMMY#3-2	0.8700	0.9886	0.9920	0.8739
DUMMY#4-1	0.9154	0.9983	0.9946	0.9698
DUMMY#4-2	0.8944	0.9968	0.9947	0.9165
DUMMY#5-1	0.8947	0.9974	0.9927	0.9747
Average	**0.9027**	**0.9964**	**0.9935**	**0.9491**

Table 6.13. YOLO performance metrics for each class and their average, achieved in experiment XII.

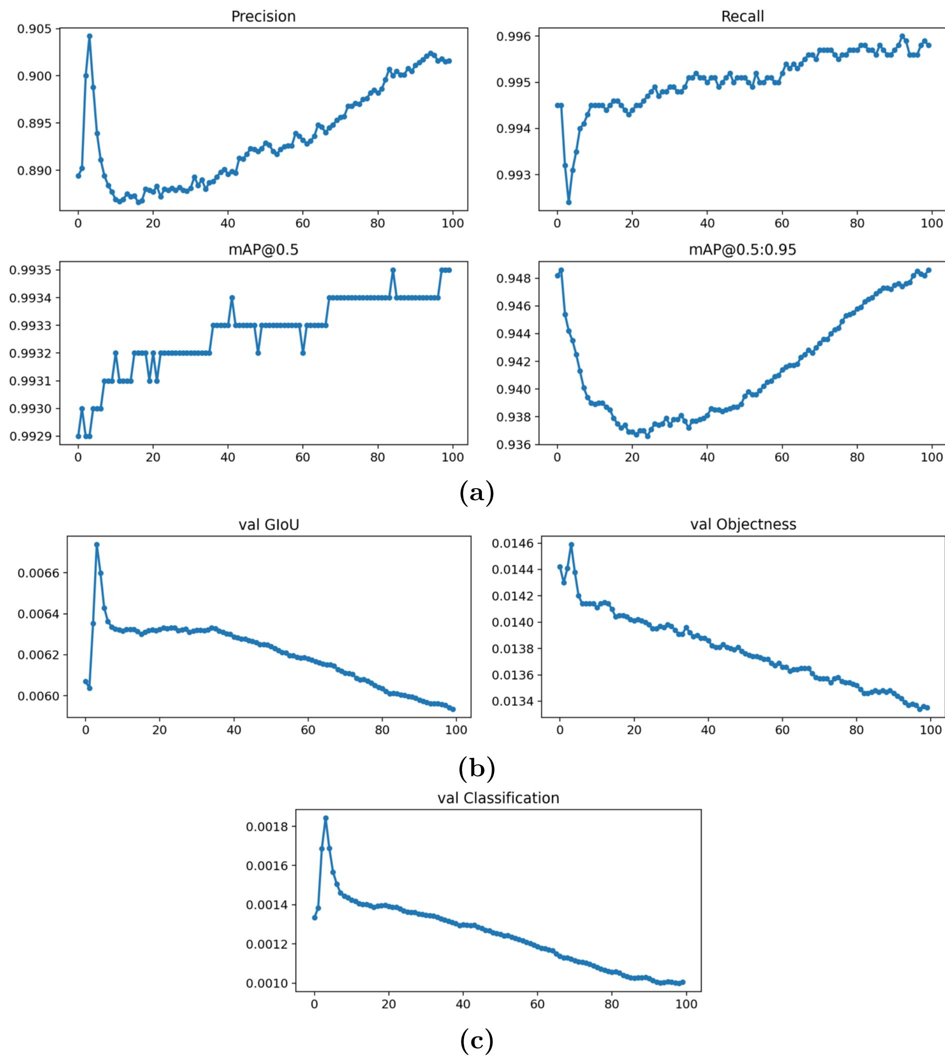


Figure 6.12. Graphs of the experiment XII representing the precision, recall, mAP 50, and mAP 95 on the test set in figure (a). Graphs representing the GIoU, objectness, and classification losses on the test set in figures (b) and (c). The trends observable in these figures are accurate: the losses tend to approach 0, while the other metrics approach 1. Although the plots may appear unusual due to the lack of straightforward improvement, a closer examination of the y-axis reveals the small magnitude of the changes. This suggests that the model has reached its peak performance, and therefore, even small fluctuations can result in significant changes in the graph.

6.2.14 Experiment XIII (PoET Training with YOLO)

In Experiment 10, we presented the PoET model training using the backbone predictions. However, since that experiment was conducted on a non-realistic dataset, once we trained the YOLO backbone in Experiment 12 using industrial textures, we also decided to train a PoET model in "backbone mode". As observed in Experiment 10, the results of this experiment are slightly worse than those achieved using the ground-truth training mode with the same dataset (i.e. Experiment 11). The achieved results are shown in Table 6.14.

If we compare the results of this experiment to those presented in Experiment 10, we can see that all the metrics are very close to each other. However, when looking at the ground-truth training results with the two datasets, Experiment 11 and Experiment 8, the PoET model trained with industrial textures performed significantly better than the one trained with non-realistic textures. Therefore, we could expect that the results of this experiment would also be better than those presented in Experiment 10.

If we examine the two graphs showing the losses, Figure 6.13 and Figure 6.9, we can see that the training in Experiment 10 ran for approximately 30 more epochs than in this experiment. This was mainly due to a lack of time. Therefore, even though the results in Table 6.14 are very similar to those in Table 6.11, we achieved these results in 30 fewer epochs. It is likely that if we had run this experiment for the same number of epochs as in Experiment 10, the results would have been better. This would also align with the trends observed in the ground-truth training results in Experiment 11 and Experiment 8.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	75.57	44.51	94.46	99.65	32.01	3.31
DUMMY#1-2	72.91	47.38	87.82	99.28	29.87	5.34
DUMMY#5-2	75.29	55.23	89.11	98.62	33.90	4.50
DUMMY#5-3	78.46	61.22	94.66	99.26	78.52	3.49
DUMMY#6-1	25.49	7.47	27.95	47.70	40.38	2.89
DUMMY#6-2	83.24	75.75	97.45	99.76	28.99	3.31
DUMMY#1-3	67.85	38.99	80.48	98.55	21.54	4.15
DUMMY#2-1	69.79	43.74	84.60	97.08	26.95	6.48
DUMMY#2-2	68.27	37.31	81.41	98.81	52.07	7.77
DUMMY#3-1	71.04	44.52	85.92	97.67	32.02	6.79
DUMMY#3-2	64.30	33.70	76.76	95.44	43.45	7.49
DUMMY#4-1	78.38	60.32	95.09	99.30	29.86	4.74
DUMMY#4-2	78.10	64.60	92.16	98.49	29.72	4.15
DUMMY#5-1	74.30	50.23	89.88	99.20	26.76	5.19
Average	**70.21**	**47.50**	**84.12**	**94.91**	**35.87**	**4.93**

Table 6.14. This table shows the updated ADD(-S), thresholds at 2%, 5%, and 10%, ARE and ATE obtained from the experiment on the testing set, along with the accuracy at different thresholds in the experiment XIII.

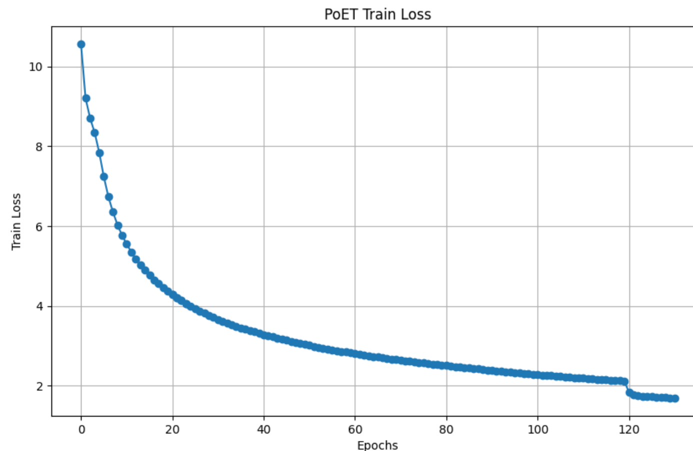


Figure 6.13. Graph showing the loss trend during the training in experiment XIII.

6.2.15 Experiment XIV (PoET Training with YOLO Fine-Tuning)

The last experiment was mainly focused on fine-tuning the model for the use case we are addressing. The model was trained by considering two main factors: accuracy and inference time. Trivially, there is a trade-off between these two parameters; the more precise the model, the higher the inference time (i.e., more time needed for each prediction). In the AR use case, precision is important because we want to correctly classify each object and determine the position to attach a 3D label for description. However, if we had to assign priority to the parameters, we should probably give more weight to inference time rather than accuracy. Thus, we adjusted the transformer hyperparameters to speed up the process without losing too much accuracy. Instead of using 5 encoding and decoding layers, we used 4 layers, which also allowed us to increase the batch size during training. We did not modify the number of heads in the transformer, as we observed that reducing this number caused performance to decrease too quickly. We used the industrial dataset with the associated pre-trained weights from the previous experiments. The results are visible in Table 6.15. The results are worse than those presented in Experiment 13, but we achieved a 10% speedup in inference time.

As mentioned, in the presented use case, inference time is a key constraint and holds more importance than accuracy. However, after achieving these results, we noticed that the performance degradation might be too significant. The ARE has increased by 5 degrees. If this error were equally distributed across the three axes, it would represent only a small increase. Nevertheless, the error distribution is not always evenly split.

Additionally, if we closely examine the ADD(-S) and the corresponding threshold ranges, we can see that the metrics have decreased by 5-6 percentage points. This, along with the increased ARE, indicates that the predicted poses are significantly worse than those achieved with the original configuration in Experiment 13. Further tests are needed to assess whether this performance degradation is still acceptable in the real-world use case.

This experiment also showed that reducing the number of encoding and decoding layers, while keeping the number of heads and other hyperparameters unchanged, makes the model significantly faster but leads to noticeable performance degradation.

Increasing the number of heads could likely help reduce this degradation, though this would also reduce the gain in inference time.

Object Name	ADD(-S)	2% T	5% T	10% T	ARE	ATE
DUMMY#1-1	70.83	40.73	89.11	95.49	37.31	4.41
DUMMY#1-2	65.91	44.19	83.35	93.28	35.44	6.26
DUMMY#5-2	69.49	50.59	82.93	91.82	38.63	4.90
DUMMY#5-3	74.14	55.15	89.98	95.12	84.14	4.44
DUMMY#6-1	20.34	3.73	22.98	40.76	45.53	3.67
DUMMY#6-2	77.01	69.42	91.41	93.25	32.97	3.98
DUMMY#1-3	62.44	32.14	74.33	93.67	25.84	4.98
DUMMY#2-1	66.07	38.01	78.02	93.07	31.05	7.20
DUMMY#2-2	62.45	30.71	74.57	93.77	56.12	8.90
DUMMY#3-1	67.50	40.24	80.79	93.10	35.59	7.88
DUMMY#3-2	60.53	29.70	71.06	88.72	47.93	8.10
DUMMY#4-1	72.93	56.99	88.79	94.47	36.16	4.98
DUMMY#4-2	71.14	57.88	88.16	91.74	35.93	4.33
DUMMY#5-1	71.26	46.47	86.51	94.55	30.39	5.61
Average	**65.15**	**42.57**	**78.71**	**89.49**	**40.93**	**5.69**

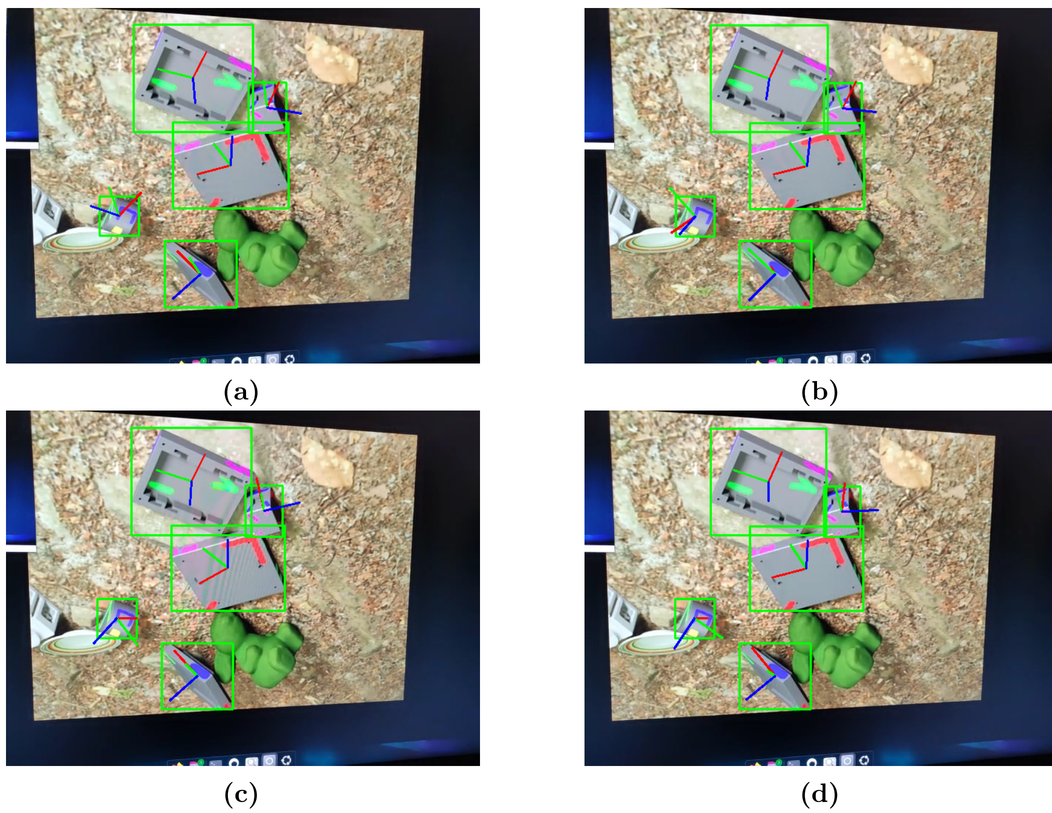
Table 6.15. This table shows the updated ADD(-S), thresholds at 2%, 5%, and 10%, ARE, and ATE for experiment XIV.

6.2.16 Inference Experiment

Once we completed all the training, we decided to test the models in inference mode. To carry out this part, we used the scripts described in Section 5.4.2. For time reasons, we used a simple webcam instead of the Varjo visor; however, this did not affect the results. Figure 6.14 shows subsequent frames taken from the inference video. To recreate the scenario, we displayed an ad-hoc image on a monitor, an image the model had never seen during training. We did not use real dummies as we lacked the time and authorization to print them using a 3D printer. The model checkpoint used was the one obtained in Experiment 10, as it was the only model trained using the YOLO backbone (Experiment 13 and Experiment 10 were completed one week before the thesis submission, so we had no time to perform the inference using those checkpoints). For real-time prediction tasks, the model must be trained with YOLO as the backbone rather than ground-truth labels. Otherwise, the images will not be pre-processed by the backbone, and PoET will not be able to predict the pose based on the backbone predictions.

As shown in Figure 6.14, the predictions are consistent across frames for most of the dummies. However, the predicted rotation for the small dummy in the lower-left corner is consistent between frames (a) and (b), but not in frames (c) and (d). The same issue is observed with the dummy at the bottom of the frame, where the prediction in frame (b) is entirely incorrect. This discrepancy is due to the model, which generally performs well but is still prone to significant wrong predictions. Overall, the model works well for most of the dummies, as reflected in the results shown in the corresponding experiment tables.

During these tests, we observed that the predictions suffer from jittering, which results in trembling axes in the videos. This issue is partly due to the unstable video but is primarily caused by the model's predictions. This issue can be mitigated by adjusting each prediction, ensuring that the rotation does not vary beyond a certain threshold. With this simple yet effective solution, the predicted pose will be more stable and less prone to oscillation.



(a) (b) (c) (d)

Figure 6.14. These are 4 subsequent frames taken from the inference video. The video was created through the inference scripts and a simple webcam by framing a monitor with frames never seen during the training.

Chapter 7

Conclusions

In the previous chapters, we discussed the problem, analyzing the solutions attempted, and the experiments conducted to achieve the initial goal of assisting the assembly line at every step. In other words, we focused on detecting and regressing each object class and pose. The dataset was the first challenge we addressed to ensure greater flexibility throughout the process. The ability to create an ad-hoc dataset in the industrial environment is fundamental. It allows us to focus on the model, rather than manually creating a new dataset for each purpose. In this project, we successfully achieved and improved the results proposed in the original paper [47]. To reach our goal, we began with the pre-trained models from the original paper and then trained and fine-tuned them on our custom dataset whenever possible.

Through the 14 experiments conducted, we tested the system using a wide range of configurations. To summarize the results, we can conclude that Experiment 13 and Experiment 14 represent the two final configurations we would adopt for the presented use case. While Experiment 13 can be considered the best in terms of accuracy, Experiment 14 is the configuration that better suits the real-time constraints imposed by the specific use case.

We demonstrated that, deep learning computer vision techniques can be applied in the Industry 4.0 environment, to solve problems in a more flexible way. However, there are still some limitations to consider. The computational complexity of the models requires high-end hardware for training and, in this specific case, also for deployment. Moreover, training models solely with synthetic data, as we did in this project, can be useful during the training and testing phases, but for a more finely tuned model, it would be beneficial to include some real-labeled images in the dataset.

In conclusion, this work represents a foundational step toward developing a flexible solution for the modern industrial environment, achieving state-of-the-art results in the 6D pose estimation task. This advancement not only enhances the accuracy and reliability of pose estimation, but also opens avenues for further research and application in dynamic industrial settings.

Chapter 8

Future Work

The project is still not entirely completed, as two minor steps remain. We developed the client for the Varjo visor; however, the predictions retrieved from the server (which runs the pose estimation model) are still not utilized. The next step is to take these predictions and render them using the custom SDK provided by Varjo. These predictions involve placing a label above each object and drawing the 3D axes representing the pose.

The project was also developed with future growth in mind. The model we used is a state-of-the-art model with top scores achieved in the BOP challenge datasets. As a result, further improvements may be difficult, as we have likely reached a plateau in model performance. However, as new models and research papers on pose estimation are likely to emerge in the coming years, we prepared for this by splitting the project into two macro-projects: the dataset generator and the pose estimation model. One potential future improvement is implementing a more advanced model when it becomes available.

One way to enhance model performance is by post-processing the predictions so that the predicted rotations during frames are influenced by the previous frame. This approach can help mitigate the prediction jittering problem described in Inference Experiment. We observed that the drawn axes tend to oscillate within small ranges and, at times, are entirely incorrect. A possible solution involves imposing constraints on the predictions across frames, ensuring that the rotations cannot vary too much between consecutive frames. This method would allow even initially incorrect predictions to be adjusted toward the correct pose in subsequent frames.

Bibliography

[1] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019.

[2] Licheng Liu, Wanli Ouyang, Xiaogang Wang, and et al. Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, 128:261–318, 2020.

[3] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[4] Lixuan Du, Rongyu Zhang, and Xiaotian Wang. Overview of two-stage object detection algorithms. *Journal of Physics: Conference Series*, 1544(1):012033, may 2020.

[5] Ross Girshick. Fast r-cnn, 2015.

[6] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[7] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection, 2017.

[8] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[10] Yifan Zhang, Xu Li, Feiyue Wang, Baoguo Wei, and Lixin Li. A comprehensive review of one-stage networks for object detection. In *2021 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, pages 1–6, 2021.

[11] Muhammad Hussain. Yolo-v1 to yolo-v8, the rise of yolo and its complementary nature toward digital manufacturing and industrial defect detection. *Machines and Tooling*, 11:677, 06 2023.

[12] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[13] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. *SSD: Single Shot MultiBox Detector*, page 21–37. Springer International Publishing, 2016.

[14] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers, 2020.

[15] Jian Guan, Yingming Hao, Qingxiao Wu, Sicong Li, and Yingjian Fang. A survey of 6dof object pose estimation methods for different application scenarios. *Sensors*, 24(4), 2024.

[16] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. Multi-view 3d object detection network for autonomous driving. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6526–6534, 2016.

[17] Štěpán Obdržálek, Gregorij Kurillo, Ferda Ofli, Ruzena Bajcsy, Edmund Seto, Holly Jimison, and Michael Pavel. Accuracy and robustness of kinect pose estimation in the context of coaching of elderly population. In *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1188–1193, 2012.

[18] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[19] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[20] Stefan Hinterstoißer, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Kurt Konolige, Gary R. Bradski, and Nassir Navab. Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes. In *Asian Conference on Computer Vision*, 2012.

[21] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, and Carsten Rother. Learning 6d object pose estimation using 3d object coordinates. In *European Conference on Computer Vision*, 2014.

[22] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. Posecnn: A convolutional neural network for 6d object pose estimation in cluttered scenes. *ArXiv*, abs/1711.00199, 2017.

[23] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, and Dieter Fox. Deepim: Deep iterative matching for 6d pose estimation. *International Journal of Computer Vision*, 128:657 – 678, 2018.

[24] Yisheng He, Wei Sun, Haibin Huang, Jianran Liu, Haoqiang Fan, and Jian Sun. Pvn3d: A deep point-wise 3d keypoints voting network for 6dof pose estimation, 2019.

[25] Nuno Pereira and Luís A. Alexandre. Maskedfusion: Mask-based 6d object pose estimation. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 71–78, 2020.

[26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *ArXiv*, abs/2010.11929, 2020.

[27] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017.

[28] Colin Rennie, Rahul Shome, Kostas E. Bekris, and Alberto F. De Souza. A dataset for improved rgbd-based object detection and pose estimation for warehouse pick-and-place. *CoRR*, abs/1509.01277, 2015.

[29] Tomás Hodan, Pavel Haluza, Stepán Obdrzálek, Jiri Matas, Manolis I. A. Lourakis, and Xenophon Zabulis. T-less: An rgb-d dataset for 6d pose estimation of texture-less objects. *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 880–888, 2017.

[30] Tomas Hodan, Frank Michel, Eric Brachmann, Wadim Kehl, Anders Glent Buch, Dirk Kraft, Bertram Drost, Joel Vidal, Stephan Ihrke, Xenophon Zabulis, Caner Sahin, Fabian Manhardt, Federico Tombari, Tae-Kyun Kim, Jiri Matas, and Carsten Rother. Bop: Benchmark for 6d object pose estimation, 2018.

[31] Roman Kaskman, Sergey Zakharov, Ivan Shugurov, and Slobodan Ilic. Homebreweddb: Rgb-d dataset for 6d pose estimation of 3d objects. *International Conference on Computer Vision (ICCV) Workshops*, 2019.

[32] Stephen Tyree, Jonathan Tremblay, Thang To, Jia Cheng, Terry Mosier, Jeffrey Smith, and Stan Birchfield. 6-dof pose estimation of household objects for robotic manipulation: An accessible dataset and benchmark. In *International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[33] Till Grenzdörffer, Martin Günther, and Joachim Hertzberg. Ycb-m: A multi-camera rgb-d dataset for object recognition and 6dof pose estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3650–3656, 2020.

[34] Benjamin Planche, Ziyan Wu, Kai Ma, Shanhui Sun, Stefan Kluckner, Terrence Chen, Andreas Hutter, Sergey Zakharov, Harald Kosch, and Jan Ernst. Depthsynth: Real-time realistic synthetic data generation from cad models for 2.5d recognition, 2017.

[35] Josip Josifovski, Matthias Kerzel, Christoph Pregizer, Lukas Posniak, and Stefan Wermter. Object detection and pose estimation based on convolutional neural networks trained with synthetic data. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6269–6276, 2018.

[36] Georgios Georgakis, Srikrishna Karanam, Ziyan Wu, and Jana Kosecka. Learning local rgb-to-cad correspondences for object pose estimation. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 8966–8975, 2018.

[37] Yuval Litvak, Armin Biess, and Aharon Bar-Hillel. Learning pose estimation for high-precision robotic assembly using simulated depth images. *2019 International Conference on Robotics and Automation (ICRA)*, pages 3521–3527, 2018.

[38] Jonathan Tremblay, Thang To, and Stan Birchfield. Falling things: A synthetic dataset for 3d object detection and pose estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[39] Chen Chen, Xin Jiang, Weiguo Zhou, and Yun-Hui Liu. Pose estimation for texture-less shiny objects in a single rgb image using synthetic training data. *arXiv preprint arXiv:1909.10270*, 2019.

[40] Tobias Löfgren and Daniel Jonsson. Generating synthetic data for evaluation and improvement of deep 6d pose estimation, 2020.

[41] Marco Ruiz, Jefferson Fontinele, Ricardo Perrone, Marcelo Santos, and Luciano Oliveira. A tool for building multi-purpose and multi-pose synthetic data sets. In *VipIMAGE 2019: Proceedings of the VII ECCOMAS Thematic Conference on Computational Vision and Medical Image Processing, October 16–18, 2019, Porto, Portugal*, pages 401–410. Springer, 2019.

[42] Artúr I Károly and Péter Galambos. Automated dataset generation with blender for deep learning-based object segmentation. In *2022 IEEE 20th Jubilee World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 000329–000334. IEEE, 2022.

[43] Apoorva Beedu, Huda Alamri, and Irfan Essa. Video based object 6d pose estimation using transformers, 2022.

[44] Ze Liu, Jia Ning, Yue Cao, Yixuan Wei, Zheng Zhang, Stephen Lin, and Han Hu. Video swin transformer. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3192–3201, 2021.

[45] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. Beit: Bert pre-training of image transformers, 2022.

[46] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[47] Thomas Georg Jantos, Mohamed Amin Hamdad, Wolfgang Granig, Stephan Weiss, and Jan Steinbrener. Poet: Pose estimation transformer for single-view, multi-object 6d pose estimation. In *Conference on Robot Learning*, pages 1060–1070. PMLR, 2023.

[48] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-YOLOv4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13029–13038, June 2021.

[49] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[50] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of cnn, 2019.

[51] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

[52] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection, 2021.