POLITECNICO DI TORINO

MASTER's Degree in MECHATRONIC ENGINEERING

MASTER's Degree Thesis

Semantic Obstacle Avoidance of Robotic Arm for Fruit Harvesting

Supervisors

Prof. Marcello CHIABERGE

PhD Mauro MARTINI

Dott. Marco AMBROSIO

Dott. Alessandro NAVONE

Candidate

Daniele TANDA

OCT 2024

Abstract

With the technological progress of robotics, Artificial Intelligence and vision systems, the projects and applications that regard the automation of simple and repetitive tasks are spreading, and, from a business point of view, business owners are more interested in automating activities for which it is difficult to find personnel. Moreover, Fruit Harvesting heavily relies on intensive human labor and sometimes it can be physically challenging. These considerations, together with the need to decrease the cost of the harvesting task (the most expensive regarding fruit agriculture), lead to the development of solutions that involve the help of automation.

Despite the efforts of the actual research in robotics, the main issue of this kind of system is that machines and robotic systems need to reach a high level of complexity in order to behave and execute a series of tasks that a human would perform unconsciously and with a certain dexterity (for example the collision avoidance of the arm with the environment).

The solution proposed in this work is an automated fruit harvesting system employing a manipulator placed on a mobile platform that moves through the orchard. The manipulator that has been used to develop this thesis is a 6-degree-of-freedom robot. The vision capabilities are guaranteed by an RGB-D camera that is used in common vision applications. To complete and enhance the capabilities of the vision system, it has been necessary to train a semantic segmentation model (based on Convolutional Neural Networks) to make sure that the environment is correctly mapped and to provide the right inputs to the system.

The system relies on a middleware, by which it is possible to build a robust logic and to develop a reliable and working application. Moreover, the usage of a platform for robot control has been fundamental for path/trajectory planning and, thanks to its planners and plugins, it has been possible to achieve a good result in terms of system reliability. The presence of built-in functions proved to be fundamental for the collision avoidance of the manipulator with the so-called hard obstacles that can cause damage to the manipulator or the plant itself.

This thesis aims to develop an automatic fruit harvesting system that can be comparable to human performances and it can work autonomously without colliding with potential sources of damage. This work intends to build a vision system that is capable of reliably recognizing the elements of the scene to allow a clear understanding of the environment. The results show that the system is highly reliable and can successfully grasp a discrete amount of apples in a relatively short time. Furthermore, the system can perceive and recreate a high-fidelity environment thanks to the high accuracy of the semantic segmentation model.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main topic of this thesis regards service robotics in the field of agriculture. In modern days there are many tasks and activities linked to fruit agriculture and extensive agriculture that are considered human-intensive and repetitive. Examples of these tasks are:

- Fruit harvesting
- Fertilization
- Pruning
- Seeding

The role of robotics must be to simplify the processes or tasks that do not require particular precision or versatility, because the machine or the robot itself must be capable of working autonomously, safely, fastly, and reliably.
The goal of this thesis is to build an autonomous robotic system for *fruit harvesting* that can guarantee strong reliability and a deep understanding of the environment and the surrounding scene.

Fruit harvesting is not a continuous task that can be done throughout all the year but it requires many people (especially for large farms) and resources. Moreover, as previously mentioned, it is highly repetitive and, sometimes, workforce demand does not match the actual supply of people that are willing to do this kind of job.

Moreover, to achieve high reliability, the robotic system must operate in raw environments that are not well-optimized for robot intervention. For this reason it is essential to prevent that the robot gets stuck or damages itself during fruit harvesting, meaning that it must consider and be aware of the surrounding environment.

Chapter 2

State of the Art

The following sections will focus on the actual state of the art. The activities and the studies that are reported in the articles cited below are mainly concerned about the vision and recognition, the grasping, and the segmentation of target fruit (that are apples in the majority of the cases).

2.1 The Monash Apple Retrieving System: A Review on System Intelligence and Apple Harvesting Performance

This paper provides a review of the Monash Apple Retrieving System, which integrates technologies such as computer vision, robotics, and artificial intelligence (AI) to improve apple harvesting efficiency [1]. The system delves in the field of automation of fruit harvesting, a task that has traditionally relied on human labor. This review discusses the system's architecture, performance metrics, and the challenges that arise when deploying such technology in real-world orchard environments.

System Architecture and Intelligence

The Monash Apple Retrieving System is designed with a modular architecture that integrates multiple subsystems, each responsible for different tasks such as fruit detection, robotic arm control, and environmental sensing. The system employs state-of-the-art computer vision algorithms to detect apples on trees. These algorithms are capable of operating under various lighting conditions and can differentiate between ripe and unripe apples based on color, size, and texture features.

A key aspect of the system's intelligence lies in its use of machine learning models that have been trained on large datasets of apple images. These models allow the system to adapt to different orchard environments by learning from new data. Additionally, the system incorporates AI-driven decision-making processes that optimize the harvesting strategy, taking into account factors such as the density of apples on a tree and the estimated ripeness of the fruit. The picking strategy and the optimization of the fruit selection are the main topics of this paper.

Harvesting Performance and Metrics

The performance of the Monash Apple Retrieving System is evaluated using several key metrics, including fruit detection accuracy, harvesting speed, and the percentage of fruit damage during harvesting. The system has demonstrated high accuracy in detecting apples, with reported detection rates exceeding 95% in controlled environments. The robotic arm, equipped with soft grippers, is designed to minimize damage to the fruit, achieving a damage rate of less than 2%. Moreover, they have managed to reach a median cycle time of 9.18 s and a 9% collision rate with the branches.

One of the notable achievements of the system is its ability to operate autonomously over extended periods, handling the variability in tree structures and fruit placement. However, the system's performance can be affected by factors such as the presence of leaves and branches that obscure the fruit, variations in tree height, and the uneven distribution of apples across the orchard.

Challenges in Real-World Deployment

Figure 2.1: Manipulator and setup used for testing

Despite its impressive capabilities, the Monash Apple Retrieving System faces several challenges in real-world deployment. Environmental variability is one of the primary challenges, as the system must be able to adapt to different orchard conditions, including varying weather patterns, tree architectures, and fruit densities. The robustness of the computer vision algorithms is crucial for maintaining high detection accuracy under these conditions. Especially because the orchard in which the system was tested was not optimized in terms of tree architecture, in order to host an automated fruit picking system.

Another challenge is the need for real-time processing and decision-making. The system must quickly process large amounts of data from multiple sensors to identify apples, plan the movement of the robotic arm and execute the harvesting action. This requires significant computational power and efficient algorithms to ensure that the system operates within the required time constraints and with the lowest possible cycle time.

Future Research Directions

Future research on the Monash Apple Retrieving System is likely to focus on improving the adaptability of the system's algorithms and hardware. Enhancements in machine learning models, particularly in the area of transfer learning, could enable the system to better generalize across different types of orchards and fruit varieties. Additionally, improvements in the design of the robotic arm and grippers could further reduce the rate of fruit damage and increase the efficiency of the harvesting process. Furthermore, some other conditions like kinematic feasibility, collision avoidance, trajectory optimality, and trajectory feasibility are not taken into account in the definition of the task and the grasping.

The integration of additional sensors, such as hyperspectral cameras, could provide more detailed information about the ripeness and health of the fruit, enabling more precise harvesting decisions. Moreover, research into energy-efficient designs could extend the operational time of the system, making it more viable for large-scale commercial use.

2.2 Sensing and Automation in Pruning of Apple Trees

Pruning is a critical task in apple orchards, essential for maintaining tree health, improving fruit quality, and maximizing yield. This paper reviews the advancements in sensing and automation technologies that are transforming the pruning process from a labor-intensive task to an automated operation [2]. The review covers various sensors used to capture the complex structure of apple trees and the algorithms

that process this data to identify optimal pruning points.

Sensing Technologies for Tree Modeling

The first step in automated pruning is the accurate modeling of the tree structure. The review highlights several sensing technologies that are used to capture 3D models of apple trees. LiDAR (Light Detection and Ranging) sensors are commonly used for this purpose, as they can produce high-resolution point clouds that represent the tree's structure. RGB-D cameras, which provide both color and depth information, are also widely used in tree modeling.

Multispectral and hyperspectral imaging sensors are increasingly being integrated into pruning systems to provide additional information about the health and vigor of the branches. These sensors can detect variations in leaf color and texture that are indicative of disease or stress, helping to identify branches that need to be pruned. Also in this case, *Computer Vision* plays a fundamental role for the identification of possible diseases or points of weakness.

Automation in Pruning

Once the tree structure is modeled, the next step is to automate the pruning process. Robotic arms equipped with cutting tools (scissors) are used to execute the pruning cuts. The paper discusses the development of algorithms that can identify the optimal pruning points by analyzing the 3D model of the tree. These algorithms take into account factors such as branch thickness, angle, and proximity to other branches.

The robotic arms are controlled by advanced motion planning algorithms that ensure precise movements, even in complex tree structures. The integration of force sensors allows the system to adjust the cutting force based on the resistance encountered, preventing damage to the tree and the cutting tool.

Challenges in Automated Pruning

Automating the pruning process presents several challenges, particularly in dealing with the variability in tree structures. Each apple tree has a unique shape, and the branches can grow in unpredictable patterns, making it difficult to apply a one-size-fits-all approach. The development of adaptive algorithms that can learn from previous pruning actions and adjust to different tree structures is an ongoing area of research.

Another challenge is the precision required in pruning. The cuts must be made at the correct location and angle to promote healthy regrowth and fruit production. Errors in pruning can lead to reduced yield and increased susceptibility to disease.

Ensuring that the automated system can achieve the same level of precision as skilled human pruners is a key focus of research.

Implications for Orchard Management

The adoption of automated pruning systems has significant implications for orchard management. These systems can reduce the reliance on seasonal labor (as well as in fruit harvesting), which is becoming increasingly scarce and expensive in many regions. Automated pruning also has the potential to improve the consistency and quality of pruning, leading to better tree health and higher yields over time.

However, the high cost of these systems remains a barrier to widespread adoption, particularly for small and medium-sized orchards. The review suggests that ongoing research into cost-effective sensor technologies and the development of more affordable robotic systems will be crucial for making automated pruning accessible to a broader range of growers.

Future Directions in Pruning Automation

Future research in the field of pruning automation is expected to focus on enhancing the intelligence of the systems. This includes the development of more advanced AI algorithms that can predict the long-term effects of different pruning strategies on tree growth and fruit production. Additionally, the integration of these systems with other automated orchard management tools, such as irrigation and fertilization systems, could lead to more holistic and optimized management practices.

2.3 Fruit Detection and Recognition Based on Deep Learning for Automatic Harvesting

The application of deep learning in agricultural automation, particularly in the detection and recognition of fruits for automatic harvesting, has seen rapid advancements in recent years. This paper provides a detailed review of the state-of-the-art deep learning models used in this domain, highlighting their strengths, limitations, and potential for future development [3].

Overview of Deep Learning Models

Deep learning models, especially Convolutional Neural Networks (CNNs), have become the cornerstone of fruit detection and recognition systems. The review discusses various CNN architectures that have been adapted for this purpose, including Faster R-CNN, YOLO (You Only Look Once), and Mask R-CNN. Each

of these models offers different trade-offs in terms of detection speed, accuracy, and computational complexity.

For instance, YOLO is known for its real-time detection capabilities, making it suitable for applications where speed is critical. However, its accuracy may be lower compared to more complex models like Faster R-CNN, which, even though result to be slower, provides higher accuracy in detecting fruits, especially in cluttered environments where fruits may be partially obscured by leaves or branches.

Training and Data Requirements

The effectiveness of deep learning models depends heavily on the quality and quantity of the training data. The paper emphasizes the importance of large, annotated datasets that cover a wide range of conditions, including different fruit types, stages of ripeness, lighting conditions, and occlusions. The creation of such datasets is a labor-intensive process, often requiring manual annotation of thousands of images.

Transfer learning is highlighted as a technique that can mitigate the data requirements by leveraging pre-trained models on related tasks. By fine-tuning these models on smaller, domain-specific datasets, it is possible to achieve high performance with less data. However, the paper notes that the success of transfer learning depends on the similarity between the pre-trained model's domain and the target application.

Challenges in Real-World Applications

Despite the promising results achieved in controlled environments, several challenges remain in deploying deep learning models for fruit detection in real-world settings. One of the main challenges is the variability in environmental conditions, such as changes in lighting, weather, and the presence of other objects that may confuse the model.

Another challenge is the real-time processing requirement for autonomous harvesting. Deep learning models, particularly those with high accuracy, can be computationally intensive, making it difficult to run them on the limited hardware available on robotic harvesting systems. The review suggests that future work should focus on optimizing these models for edge computing devices, which can process data locally without relying on cloud-based resources.

Impact on Agricultural Practices

The adoption of deep learning-based fruit detection systems has the potential to revolutionize agricultural practices. These systems can significantly reduce the need

for manual labor in harvesting, leading to lower operational costs and increased efficiency. Additionally, by enabling more precise harvesting, these systems can reduce waste and improve the overall quality of the harvested produce.

However, the paper also points out that the widespread adoption of these technologies will require significant investment in infrastructure, including the development of robust, real-time processing systems and the creation of large, diverse datasets for training. Moreover, the integration of these systems with existing agricultural practices will require careful planning to ensure that they complement rather than disrupt traditional farming methods.

Future Research Directions

The future of deep learning in automatic fruit harvesting lies in the development of more robust models that can operate reliably in diverse and challenging environments. This includes research into new neural network architectures that can handle the high variability found in outdoor settings, as well as the development of self-learning systems that can adapt to new conditions without requiring extensive re-training.

Additionally, there is a growing interest in combining deep learning with other sensing modalities, such as hyperspectral imaging and LiDAR, to improve the accuracy and robustness of fruit detection systems. The integration of these technologies could lead to more comprehensive systems that not only detect and recognize fruits but also assess their quality and readiness for harvest.

2.4 Cross-Domain Matching for Semantic Point Cloud Segmentation Based on Image Segmentation and Geometric Reasoning

Semantic point cloud segmentation is a critical task in various applications, including autonomous driving, robotics, and environmental monitoring. This paper introduces a novel approach to cross-domain matching for semantic point cloud segmentation, which leverages both image segmentation techniques and geometric reasoning [4].

Cross-Domain Matching Techniques

The approach discussed in the paper involves using image segmentation as a preliminary step to guide the segmentation of 3D point clouds. By aligning the segmented image data with the 3D point cloud, the method can more accurately classify points in the cloud, even in cases where the point cloud data alone is insufficient for accurate segmentation.

Geometric reasoning is employed to refine the segmentation results, taking into account the spatial relationships between points and the overall structure of the environment. This dual approach allows for better handling of complex scenes where traditional point cloud segmentation methods might struggle.

Challenges and Limitations

One of the key challenges in cross-domain matching is the difference in data representation between images and point clouds. Images are 2D projections, while point clouds represent 3D spatial data. Aligning these two types of data requires sophisticated algorithms that can accurately map the 2D image features onto the 3D point cloud.

The paper also discusses the issue of domain adaptation. Models trained on one type of data (e.g., images) may not perform well when applied to another type (e.g., point clouds) due to differences in data distribution. The proposed method addresses this challenge by using geometric reasoning to bridge the gap between the domains, but the authors acknowledge that further improvements are needed to achieve seamless domain adaptation. Geometric reasoning is performed employing the definition of some geometric parameters that can be extracted from the point cloud and from adjacent points. In particular, objects can be extracted by clustering points based on some variables (e.g. verticality, curvature, planarity, etc ...), by means of a *region growing* algorithm.

Applications and Real-World Impact

The proposed cross-domain matching technique has significant potential for improving the accuracy and efficiency of semantic point cloud segmentation in various applications. In autonomous driving, for example, accurate segmentation of point clouds is essential for understanding the environment and making safe driving decisions. The integration of image data can enhance the segmentation of complex scenes, such as urban environments with many overlapping objects.

In robotics, the technique can be used to improve object recognition and scene understanding, enabling robots to navigate and interact with their environment more effectively. The method's ability to handle complex, unstructured environments makes it particularly useful for tasks such as search and rescue, where accurate environmental modeling is critical.

Future Research Directions

The paper suggests several avenues for future research, including the development of more advanced alignment techniques that can better integrate image and point

cloud data. Additionally, there is a need for more robust domain adaptation methods that can ensure the transferability of models across different types of data and environments.

Another promising direction is the integration of additional data modalities, such as thermal imaging or radar, which could provide complementary information to further enhance segmentation accuracy. The combination of multiple data sources could lead to the development of more comprehensive and reliable systems for semantic segmentation in a wide range of applications.

Chapter 3

Theoretical Overview

3.1 General Overview of Computer Vision

3.1.1 Introduction to Computer Vision

Computer vision is a field in artificial intelligence (AI) that enables machines to interpret and act on visual data. From early edge detection algorithms to modern deep learning models, the field has seen exponential growth, fueled by advances in computational power, data availability, and algorithmic innovations.

The evolution of computer vision spans over five decades, beginning with basic image processing techniques in the 1960s and 1970s. Early work focused on low-level tasks such as edge detection and pattern recognition. The introduction of machine learning in the 1990s, followed by deep learning in the 2010s, catalyzed a significant leap in the capabilities of computer vision systems.

3.1.2 Fundamentals of Image Processing

Image processing serves as the foundation of computer vision, involving operations that transform raw visual data into a more analyzable form. Techniques such as filtering, edge detection, and segmentation are pre-processing steps that prepare images for further analysis by machine learning models.

Filtering techniques, such as Gaussian blur and sharpening, are used to enhance or suppress specific features in an image. These operations are crucial in reducing noise and highlighting important details before further processing.

Edge Detection

Edge detection algorithms, like the Sobel [5] and Canny [6] operators, identify significant transitions in intensity, which correspond to object boundaries. These

edges provide critical information for subsequent stages of computer vision pipelines, such as feature extraction and object detection.

3.1.3 Deep Neural Networks and CNNs

Deep Neural Networks (DNNs) have transformed computer vision by enabling the automatic extraction of hierarchical features from raw images. Convolutional Neural Networks(CNNs), a specialized type of DNN, are particularly effective in visual data analysis due to their ability to learn spatial hierarchies of features.

They are made up of the following elements (schematized in figure 3.1):

- **Convolutional Layers**: Convolutional layers apply a series of learnable filters to the input image, producing feature maps that highlight various aspects of the image, such as edges, textures, and shapes. These filters are trained to recognize patterns that are critical for distinguishing between different objects.

- **Pooling Layers**: Pooling layers reduce the spatial dimensions of the feature maps, preserving the most important information while reducing computational complexity. Max pooling selects the highest value in each region of the feature map, while average pooling computes the mean value.

- **Fully Connected Layers**: Fully connected layers in a CNN aggregate the extracted features to make final predictions. Each neuron in these layers is connected to every neuron in the previous layer, allowing the network to consider all features when making a decision.

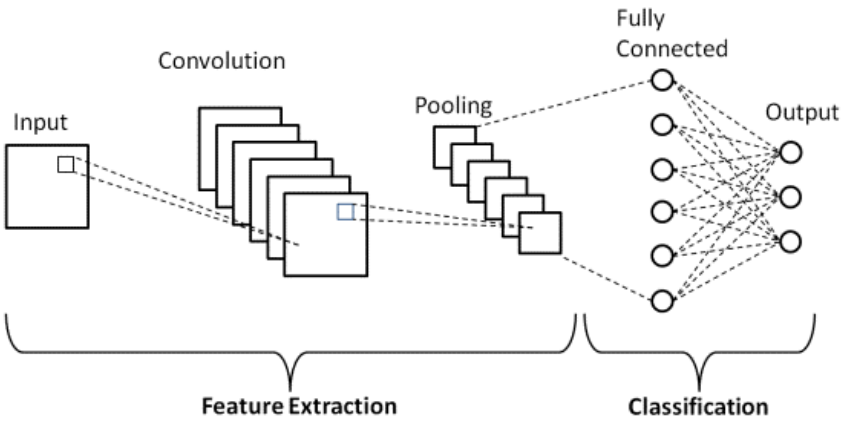

Figure 3.1: Structure of Convolutional Neural Networks [7]

3.1.4 R-CNN: Region-based Convolutional Neural Networks

Region-based Convolutional Neural Networks (R-CNNs) revolutionized object detection by combining region proposals with CNNs for classification. This approach significantly improved the accuracy of object detection but introduced challenges in terms of computational efficiency (requires a greater effort compared to one-shot models like YOLO).

R-CNN uses Selective Search to generate region proposals by combining the strengths of exhaustive search and segmentation. This method balances the trade-off between computational efficiency and the quality of region proposals.

Training R-CNN involves fine-tuning a CNN on each region proposal, followed by Support Vector Machine (SVM) training for classification and bounding box regression. This multi-stage training process is computationally expensive and time-consuming, leading to faster variants' development.

Advantages and Limitations of R-CNN

While R-CNN achieved state-of-the-art accuracy at the time of its introduction, the model's complexity and slow inference speed limited its practical applications, especially in real-time scenarios.

3.2 Introduction to YOLO

The YOLO (You Only Look Once) model introduced a new paradigm in object detection by framing the task as a single regression problem. Unlike R-CNN, which processes region proposals separately, YOLO predicts bounding boxes and class probabilities directly from the entire image in a single forward pass.

YOLO's primary innovation is its ability to perform object detection in a single evaluation, significantly speeding up the process while maintaining competitive accuracy. This efficiency makes YOLO particularly well-suited for real-time applications such as autonomous vehicles, video surveillance, or, as in the work proposed in this thesis, fruit harvesting.

The basic structure of YOLO is reported in the figure 3.2, in which it is possible to have a clear idea of how the output of the Convolutional module is managed to obtain the final output of the detection. In particular, the output block is made up of the following information:

- **X**: is in the range between -1 and 1 and it represents the X coordinate of the center of the bounding box with respect to the center of the image (must be multiplied by W/2, where W is the width of the image expressed in pixels). This parameter is predicted thanks to a *tanh* activation function.

- **Y**: is in the range between -1 and 1 and it represents the Y coordinate of the center of the bounding box with respect to the center of the image (must be multiplied by H/2, where H is the width of the image expressed in pixels). This parameter is predicted thanks to a *tanh* activation function.
- **w**: is in the range between 0 and 1 and represents the width of the bounding box in pixels (it represents a fraction of the total pixel width of the image, W). This parameter is predicted thanks to a *sigmoid* activation function.
- **h**: is in the range between 0 and 1 and represents the height of the bounding box in pixels (it represents a fraction of the total pixel height of the image, H). This parameter is predicted thanks to a *sigmoid* activation function.
- **C**: Confidence level associated with the position of the bounding box. This parameter is predicted thanks to a *sigmoid* activation function.
- **Softmax classes**: it contains the confidence level associated with each class that the model can detect. There can be the additional constraint that the sum of the confidences of all the classes must be equal to 1. Every confidence level is calculated through a *Softmax* activation function.

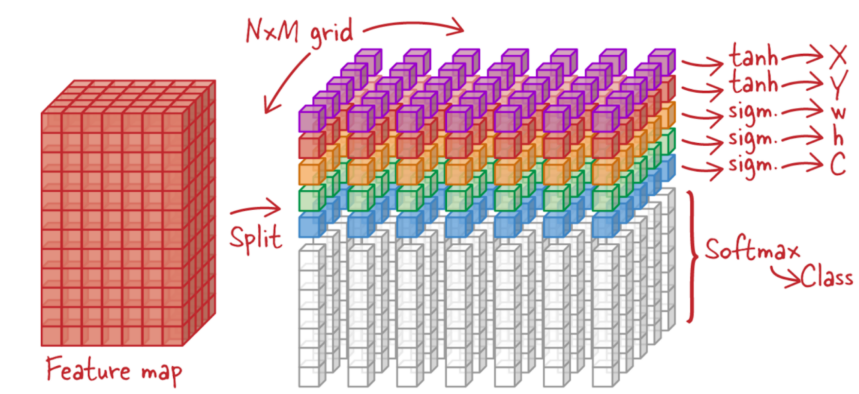

Figure 3.2: YOLO basic structure [8]

The following section will delve into a description of the YOLO models that have been released in recent years. For each model, there will be a short description that highlights the differences and the features that have been added [9].

3.2.1 YOLO: The First Version (YOLOv1)

YOLOv1 represented a significant shift in object detection by proposing a unified model that could predict multiple bounding boxes and class probabilities simultaneously. The model divides the image into an $S \times S$ grid, with each grid cell

predicting a fixed number of bounding boxes and associated class probabilities, similar to what is described in figure 3.2.

The YOLOv1 architecture comprises 24 convolutional layers followed by 2 fully connected layers. The final layer predicts both the bounding boxes and class probabilities for each grid cell. The model's simplicity contributes to its speed, but it also results in limitations, particularly in detecting small objects and handling overlapping bounding boxes.

Training Process and Loss Function

YOLOv1 uses a custom loss function that penalizes incorrect predictions of bounding boxes and class probabilities. The loss function is designed to balance localization and classification errors, but the model often struggles with the trade-off, leading to challenges in accurately detecting small or densely packed objects.

Strengths and Weaknesses

YOLOv1's main strength lies in its speed, making it suitable for real-time applications. However, its relatively low accuracy, especially in complex scenes, highlighted the need for further improvements, which were addressed in subsequent versions.

3.2.2 YOLOv2 and YOLOv3: Evolution and Improvements

YOLOv2, also known as YOLO9000, introduced several key improvements over YOLOv1, such as batch normalization, anchor boxes, and a higher-resolution classifier. These enhancements significantly improved the model's accuracy and robustness, making YOLOv2 a more versatile object detection model.

The YOLOv2 architecture features 19 convolutional layers followed by 5 max-pooling layers (represent a filter like the one that is present in convolutional layers but it returns the maximum value of the pixels present in the filter size) and a fully connected layer. The introduction of anchor boxes allowed YOLOv2 to handle objects of varying aspect ratios, addressing one of the primary limitations of YOLOv1. Anchor boxes are prediction boxes with a predefined width/height ratio that are used to make faster predictions and to avoid the problem of two overlapping bounding boxes.

Multi-Scale Training and Other Techniques

YOLOv2 introduced multi-scale training, where the model was trained on images of different scales, enhancing its ability to detect objects of various sizes. Additionally, the use of batch normalization reduced the risk of overfitting and improved convergence during training. Regarding batch normalization, one of the problems in the

learning phase of a Deep Neural Network is the fact that the range of the input data doesn't match the activation dynamic area on the activation function. This is a problem because if a lot of data are not in the dynamic area of the activation function, the optimizer can't calculate a valuable gradient to move and change the weight in a more suitable optimal region. For this reason is important to make sure that the data is normalized (by computing means and standard deviations) to obtain a suitable range of input data.

YOLOv3 and Feature Pyramid Networks

YOLOv3 built upon the foundation of YOLOv2 by incorporating residual connections and feature pyramid networks (FPNs), enabling the model to detect objects at multiple scales more effectively. The deeper architecture and multi-scale predictions made YOLOv3 one of the most accurate real-time object detectors available at the time.

YOLOv3 introduced a deeper network architecture with 53 convolutional layers, known as Darknet-53 [10]. This architecture utilizes residual connections, which allow the network to learn more complex features without suffering from the vanishing gradient problem. The model also predicts objects at three different scales, improving its ability to detect both small and large objects in an image.

3.2.3 YOLOv4 and YOLOv5: Advanced Techniques

YOLOv4 introduced several cutting-edge techniques to further enhance object detection performance, including Cross-Stage Partial connections (CSP), the Mish activation function, and Self-Adversarial Training (SAT). These innovations allowed YOLOv4 to achieve state-of-the-art performance on several benchmarks.

The YOLOv4 architecture features CSPDarknet53 [10] as the backbone, which improves gradient flow and reduces the computational burden. The use of the Mish activation function helps in stabilizing the training process, while SAT introduces adversarial examples to make the model more robust to variations in input data.

YOLOv5: Community-Driven Development

Although not officially released by the original YOLO authors, YOLOv5 became popular due to its ease of use and community-driven development. YOLOv5 is written in PyTorch and introduces several incremental improvements, such as auto-learning anchors and improved augmentation techniques.

YOLOv4 and YOLOv5 outperform many contemporary models in terms of speed and accuracy. YOLOv4, in particular, achieves high mAP (mean Average Precision) scores while maintaining fast inference times, making it suitable for both real-time and high-accuracy applications.

3.2.4 YOLOv6 and YOLOv7

YOLOv6 and YOLOv7 continue to push the boundaries of object detection by focusing on specific use cases, such as low-power devices and high-resolution image analysis.

YOLOv6 is designed for deployment on mobile and edge devices, featuring a pruned version of CSPDarknet and a novel attention mechanism. The model achieves a balance between computational efficiency and accuracy, making it ideal for resource-constrained environments.

YOLOv7 targets high-resolution object detection tasks, such as aerial imagery and medical image analysis. The model uses a modified version of PANet [11] to aggregate features from different layers, creating faster "paths" for feature extraction and enabling the detection of small objects with high precision.

3.2.5 YOLOv8: The Latest Iteration

YOLOv8 represents the pinnacle of the YOLO series, introducing a modular and scalable architecture that can be customized for a wide range of computer vision tasks.

YOLOv8 uses CSPDarknet as the backbone, PANet for path aggregation, and a flexible detection head. The modular design allows YOLOv8 to be tailored to specific tasks, from real-time object detection to large-scale image analysis.

YOLOv8 incorporates advanced techniques such as AutoAugment for data augmentation, knowledge distillation for model compression, and hybrid task learning to improve generalization. The loss function is designed to balance the trade-offs between localization, classification, and confidence, resulting in improved accuracy and robustness.

Performance Metrics

YOLOv8 achieves state-of-the-art performance on several benchmarks, with high mAP scores and fast inference times. The model is capable of real-time object detection, making it suitable for applications where both speed and accuracy are critical.

3.3 Detailed Comparison of YOLOv8 with R-CNN and Fast R-CNN

3.3.1 Architectural Differences

The primary difference between YOLOv8 and R-CNN models lies in their approach to object detection. YOLOv8 uses a single-stage detector that processes the entire image in one forward pass, while R-CNN models rely on region proposals and multiple passes through a CNN.

R-CNN models, including Fast R-CNN, use a multi-stage pipeline that involves region proposal generation, feature extraction, and classification. In contrast, YOLOv8 employs a unified architecture with CSPDarknet as the backbone, PANet for feature aggregation, and a customized detection head for predicting bounding boxes and class probabilities.

The single-stage approach of YOLOv8 provides a significant advantage in terms of speed, making it more suitable for real-time applications. R-CNN models, despite their accuracy, require more computational resources and longer inference times due to their multi-stage nature.

3.3.2 Comparison of Performance Metrcs

Table 3.1: Comparison of YOLOv8, R-CNN, and Fast R-CNN Models

Metric	YOLOv8	R-CNN	Fast R-CNN
mAP	55.3%	66.0%	70.4%
IoU	0.45	0.50	0.55
Inference Time	2.0 ms/image	47.0 ms/image	25.0 ms/image
FPS	50	2	4
Model Size	70 MB	240 MB	200 MB

The performances of the different model result to be slightly different and they are highlighted by the following performance metrics:

- **Mean Average Precision (mAP)**: mAP is a critical metric for evaluating object detection models, representing the average of the maximum precisions at different recall levels. While R-CNN models typically achieve higher mAP scores due to their region proposal mechanism, YOLOv8's performance is optimized for speed without a significant loss in accuracy.
- **Intersection over Union (IoU)**: IoU measures the overlap between the predicted bounding box and the ground truth. Higher IoU values indicate

better localization accuracy. YOLOv8 achieves a competitive IoU but is slightly lower than that of R-CNN models due to its focus on speed.

- **Inference Speed and Model Size**: Inference speed is a major advantage of YOLOv8, making it ideal for real-time applications. The model size is also significantly smaller than that of R-CNN models, which is beneficial for deployment on devices with limited resources.

3.3.3 Real-World Applications

The strengths and weaknesses of YOLOv8 and R-CNN models make them suitable for different applications. YOLOv8 excels in real-time object detection tasks, such as video surveillance and autonomous driving, where speed is critical. In the case of fruit harvesting, speed is an essential performance metric that can heavily affect cycle time and grasping rate. In contrast, R-CNN models are better suited for applications requiring high accuracy, such as medical image analysis and detailed scene understanding.

Chapter 4

Hardware and Software Intrumentation

This chapter provides a detailed examination of the tools and instrumentation utilized in this research, focusing on their roles, integration, and impact on the development of an automated fruit harvesting system. The key technologies discussed include the Robot Operating System 2 (ROS2) Humble distribution, the YOLOv8 Python library for object detection and segmentation, the Kinova Gen3 Lite robotic arm, and the Intel RealSense D435i 3D camera. Each section explores the technical details, and functionalities, along with their integration within the ROS2 environment.

4.1 ROS2 Humble

ROS2 represents the next generation of the Robot Operating System, addressing many of the limitations of ROS1, particularly in the realms of real-time performance, distributed computing, and security. The Humble Hawksbill release, being a Long-Term Support (LTS) version, offers stability and sustained support, making it an attractive option for long-term projects, especially in research and industrial applications.

4.1.1 ROS2 Architecture

The architecture of ROS2 [12] is designed around the principles of modularity and scalability, facilitating the development of complex robotic systems through a network of loosely coupled nodes. Each node operates independently, communicating through a middleware layer based on the Data Distribution Service (DDS)

standard. This decentralized approach enhances the flexibility and resilience of robotic systems, making them adaptable to various operating environments.

Key Components of ROS2

- **Nodes**: The fundamental building blocks in ROS2, representing distinct processes that perform specific tasks.
- **Topics**: Channels through which nodes communicate by publishing and subscribing to messages. Topics are strongly typed and support asynchronous communication.
- **Services**: Provide synchronous communication between nodes, allowing a node to send a request and wait for a response.
- **Actions**: An extension of services that handle long-running tasks, where feedback and preemption are necessary.
- **Parameters**: Allow dynamic reconfiguration of node settings at runtime, enhancing the adaptability of the system.

4.1.2 Node Communication and Middleware

In ROS2, communication between nodes is facilitated by the underlying DDS middleware, which supports several Quality of Service (QoS) policies. These policies are crucial for ensuring that the system meets the specific performance requirements of different applications. The following are some of the key QoS policies in ROS2:

- **Reliability**: Ensures message delivery with either best-effort or reliable modes. Reliable mode guarantees message delivery, which is essential for critical data such as control commands or sensor readings.
- **Durability**: Controls whether a topic's history is preserved for future subscribers. This is particularly useful in scenarios where new nodes join a running system and need to access previous messages.
- **Liveliness**: Monitors the availability of nodes, ensuring that a node is still active and able to communicate. This feature is crucial for maintaining the integrity of the system in dynamic environments.
- **Latency Budget**: Defines the maximum acceptable delay between message production and delivery. This is vital for applications requiring real-time performance, such as robotic vision or control systems.

4.1.3 Advanced ROS2 Concepts

Lifecycle nodes in ROS2 provide a or managing the states of a node. This feature allows developers to design nodes that can be initialized, activated, deactivated, and shut down in a controlled manner. Lifecycle management is particularly useful in large systems where resources need to be managed efficiently, and where nodes may need to be brought online or offline based on the system's operational state.

- **Inactive State**: In this state, the node is initialized but not performing any operations. This allows for resource allocation without executing the node's primary functionality.
- **Active State**: The node is fully operational and executing its tasks. Transitioning to this state from the inactive state typically involves setting up necessary publishers, subscribers, and other resources.
- **Finalized State**: The node releases all resources and shuts down, either gracefully or forcefully, depending on the system's requirements.

4.1.4 Integration with Hardware

The integration of ROS2 with hardware is facilitated by its hardware abstraction layers (HALs) and device drivers. ROS2 provides packages and plugins that allow seamless communication with various sensors, actuators, and other hardware components. For example, the integration of the Kinova Gen3 Lite robotic arm and the RealSense D435i camera is straightforward due to the availability of ROS2 drivers and packages specifically designed for these devices.

Hardware Abstraction Layers (HAL)

HALs in ROS2 provide a uniform interface to different types of hardware, allowing developers to write device-agnostic code. This is particularly useful in multi-robot systems or when switching between different hardware configurations.

ROS2 Drivers and Plugins

ROS2 supports a wide range of drivers and plugins that enable communication with hardware devices. These drivers abstract the low-level details of the hardware, allowing for high-level control through ROS2 nodes. For example, the *kinova_robots* package provides drivers for controlling Kinova robotic arms, while the *realsense*2*_camera* package offers drivers for Intel RealSense cameras.

4.1.5 Comparison with ROS1

While ROS1 laid the foundation for robotics middleware, ROS2 introduced several improvements that addressed the limitations of its predecessor. A comparison of key features between ROS1 and ROS2 includes:

- **Real-Time Support**: ROS2 is designed with real-time performance in mind, offering features like deterministic execution and priority-based scheduling.
- **Scalability**: ROS2's decentralized architecture allows it to scale better in large, distributed systems, unlike ROS1, which relies on a central master node.
- **Cross-Platform Compatibility**: ROS2 supports multiple operating systems, including Linux, Windows, and macOS, whereas ROS1 is primarily Linux-based.
- **Security**: ROS2 includes built-in security features, whereas ROS1 requires external tools and workarounds to secure communication.

4.2 YOLOv8 Python Library

The YOLOv8 (You Only Look Once) [9] object detection library is an evolution of the YOLO series, focusing on both speed and accuracy. YOLOv8 improves upon its predecessors by incorporating advancements in neural network architectures and training techniques, making it highly suitable for real-time applications such as robotic vision.

YOLOv8 introduces several architectural changes (as described in section 3.2.5) that enhance its performance. These include the adoption of a CSPNet (Cross Stage Partial Network) backbone, which reduces the computational cost without compromising accuracy. Additionally, YOLOv8 uses a PANet (Path Aggregation Network) for better feature fusion, allowing the network to capture multi-scale features more effectively.

4.2.1 API and Functionalities

YOLOv8's Python API is designed for ease of use, allowing developers to quickly load pre-trained models, perform inference, and fine-tune models on custom datasets. The API supports various functionalities, including object detection, image classification, and segmentation.

Object Detection API

The object detection API in YOLOv8 allows users to detect objects in images or video streams with minimal setup. Key parameters include:

- **Confidence Threshold**: Sets the minimum confidence score for an object to be considered detected. A higher threshold reduces false positives but may miss some true positives.
- **IoU Threshold**: The Intersection over Union (IoU) threshold determines how overlapping bounding boxes are handled. Higher IoU thresholds lead to more conservative detection, reducing false positives.
- **Multi-Scale Detection**: YOLOv8 supports multi-scale detection, enabling the model to detect objects at varying scales within the same image.

Segmentation API

The segmentation API in YOLOv8 is a powerful tool for pixel-level object recognition. It supports various training configurations and hyperparameters that can be fine-tuned to improve model performance on specific tasks.

- **Loss Functions**: The segmentation model uses a combination of binary cross-entropy loss and Dice loss to balance pixel-wise accuracy with overall segmentation quality.
- **Learning Rate Schedulers**: YOLOv8 provides various learning rate schedulers, such as step decay, to optimize training performance.
- **Data Augmentation**: Advanced data augmentation techniques, including random cropping, flipping, and color adjustments, are supported to enhance model generalization.

4.2.2 Training a Segmentation Model

Training a segmentation model in YOLOv8 involves several steps, including data preparation, model configuration, and training. Key parameters that can be configured include:

- **Batch Size**: The number of samples processed before the model's internal parameters are updated. Larger batch sizes can lead to faster training but require more memory and more performing hardware.
- **Epochs**: The number of complete passes through the training dataset. More epochs allow the model to learn better but increase the risk of overfitting.

- **Optimizer**: YOLOv8 supports optimizers such as SGD (Stochastic Gradient Descent) and Adam, which can be selected based on the specific characteristics of the dataset.
- **Input Image Size**: The input size can be adjusted to balance between computational efficiency and detection accuracy. Larger input sizes provide more detail but require more processing power.

More hyper-parameters, linked both to model training optimization and data augmentation, will be discussed and explained in section 6.1.1.

4.3 Kinova Gen3 Lite Robotic Arm

4.3.1 Mechanical Description

The Kinova Gen3 Lite [13] is a lightweight, modular robotic arm designed for versatility and ease of use. It features six degrees of freedom (DoF), providing a balance between flexibility and simplicity, making it ideal for tasks such as fruit harvesting.

Each joint of the Gen3 Lite is actuated by a brushless DC motor with integrated encoders, offering precise control and feedback. The arm's joints are configured in a serial kinematic chain, which allows it to reach a wide range of positions and orientations within its workspace.

Figure 4.1: Kinova Gen. 3 Lite Manipulator

Payload and Reach

The Gen3 Lite can handle payloads of up to 2 kg with a maximum reach of 740 mm (workspace limitation). This makes it suitable for handling small to medium-sized fruits, particularly in environments where space is constrained, such as dense orchards.

End-Effector Options

The robotic arm is compatible with various end-effectors, including grippers, suction cups, and custom tools. For fruit harvesting, a soft gripper end-effector is often preferred to minimize the risk of bruising or damaging the fruit. The end-effector can be easily swapped out depending on the specific requirements of the task. During the tests, and the development of this work, the default hard end-effector has been used (shown in figure 4.1 and in figure 4.3).

4.3.2 Control System and Software Integration

The control system of the Kinova Gen3 Lite is based on a real-time operating system (RTOS) that ensures precise timing and coordination of movements. The arm can be controlled via ROS2, with the *kinova_robots* package providing the necessary drivers and interfaces for communication between ROS2 nodes and the robot's control system.

Kinematic Control

The Gen3 Lite supports both forward and inverse kinematics, enabling it to calculate the necessary joint angles to reach a desired position and orientation. The inverse kinematics algorithms are optimized for speed and accuracy, making the arm responsive to real-time changes in its environment, such as the movement of fruit on a tree branch.

Motion planning for the Gen3 Lite is handled by the MoveIt 2 framework [14], which integrates seamlessly with ROS2. MoveIt allows for the generation of collision-free trajectories, which is essential when operating in cluttered environments like orchards. The planning algorithms, later discussed in section 5.3, take into account the robot's kinematics, dynamic constraints, and the presence of obstacles, ensuring safe and efficient operation.

4.3.3 Suitability for Fruit Harvesting

The Kinova Gen3 Lite's combination of precision, modularity, and ease of integration makes it an excellent choice for fruit harvesting tasks. The arm's lightweight design

allows it to be mounted on mobile platforms or within constrained spaces, while its advanced control algorithms ensure that it can delicately handle fruits without causing damage. This is guaranteed by the fact that the fingers of the gripper can be controlled, not just by giving a target pose, but also by the maximum effort that can be exerted by the motors of the gripper.

Energy Efficiency

The energy efficiency of the Gen3 Lite is another important factor, particularly for mobile platforms that rely on battery power. The arm's low power consumption, combined with its ability to enter a low-power state when not in use, extends the operational time of the mobile platform, allowing for longer periods of autonomous operation in the field.

4.4 3D Cameras in Robotic Vision

3D cameras are essential components in robotic vision systems, providing depth information that is critical for tasks such as object recognition, navigation, and manipulation. Various 3D sensing technologies are available, including stereo vision, structured light, and time-of-flight (ToF).

Stereo Vision

Stereo vision systems use two or more cameras to capture images from slightly different perspectives. By comparing these images, the system can calculate the distance to objects based on the disparity between the images. This method is effective for outdoor environments where ambient light is available, but it can be less reliable in low-light conditions.

Structured Light

Structured light systems project a known pattern (such as a grid or dot matrix) onto a scene and then capture the deformation of this pattern with a camera. The deformation provides information about the depth of the objects in the scene. Structured light is often used in controlled environments where lighting conditions can be managed.

Time-of-Flight (ToF)

ToF cameras emit a light pulse (usually infrared) and measure the time it takes for the light to reflect back to the sensor. The time delay corresponds to the distance of

the object from the camera. ToF cameras are highly effective in indoor environments and can provide accurate depth information even in low-light conditions.

4.4.1 Intel RealSense D435i

The Intel RealSense D435i [15] is a stereo-based 3D camera equipped with an integrated IMU, making it suitable for a wide range of robotic applications. The camera combines depth sensing, color imaging, and inertial measurement in a compact form factor.

Figure 4.2: Intel@ RealSense D435i

Technical Specifications

The following table provides a sufficient overview of the technical specifications of the camera:

General information	
External dimensions ($L \times W \times H$)	90 × 25 × 25 mm
Ideal Range	0.3 to 3 m
Use environment	Indoor/Outdoor
Depth	
Depth technology	Stereoscopic
FOV	87° × 58°
Resolution	Up to 1280 × 720
Frame rate	Up to 90 fps
Depth Accuracy	$<$ 2% at 2 m
Minimum Depth Distance at Max Resolution	28 cm
RGB	
RGB technology	Rolling shutter
FOV	69° × 42°
Resolution	Up to 1920 × 1080
Frame rate	Up to 30 fps
Sensor resolution	2MP

Table 4.1: Technical specifications of Intel Realsense Depth Camera D435i

Figure 4.3: Setup of Kinova Gen. 3 Lite Manipulator with Intel RealSense D435i

Depth Sensing Capabilities

The D435i utilizes a pair of infrared cameras to capture stereo images, which are then processed to generate a depth map. The camera's ability to produce accurate depth information even in challenging lighting conditions makes it well-suited for outdoor applications, such as fruit harvesting, where lighting can vary significantly.

The camera has been mounted on the `gripper_base_link` as shown in figure 4.3. This choice is fundamental because thanks to this setup the camera can always provide an unobstructed view of the environment and it can guarantee clear and high-resolution frames.

4.4.2 Integration with ROS2

The integration of the D435i with ROS2 is facilitated by the *realsense2_camera* package, which supports various ROS2 topics and services for accessing the camera's data. The package provides topics for depth images, color images, and IMU data, as well as services for camera calibration and configuration.

Depth Image Processing

The depth images from the D435i can be processed using various ROS2 packages, such as *image_pipeline* and *pointcloud_to_laserscan*, to extract relevant features for object detection and manipulation. The fundamental and most important function is *rs2_deproject_pixel_to_point*, which is a method of `pyrealsense` library that is entitled to project a pixel into a point in the 3D space as explained later in section 5.2.5. For example, in a fruit harvesting task, the depth data can be used to calculate the distance to the fruit and guide the robotic arm to the correct position.

Visual-Inertial Odometry

The IMU data from the D435i can be combined with the depth data to perform visual-inertial odometry (VIO), which is crucial for navigating the robot through complex environments. The *realsense2_camera* package provides built-in support for VIO, making it easier to implement advanced navigation algorithms in ROS2.

Chapter 5

System Description and Implementation

The following chapter is dedicated to the explanation of the system's working principle and how the work has been structured in order to build the autonomous system for fruit harvesting. The system's development has been divided into 3 parts to make it simpler and to have the possibility to develop each part separately. After the development of all the parts, an integration test must be performed in order to state and determine whether all the parts are compatible and the interfaces work properly.

The three parts are the following:

- **Computer Vision**
- **Sensor Matching**
- **Trajectory Planning, Obstacle Avoidance and Grasping**

5.1 Computer Vision Task

The main goal of this part of the system is to easily recognize the **target fruit**, **hard obstacles** and **soft obstacles**.

The **hard obstacles** are defined as those obstacles that need to be absolutely avoided during the trajectory planning, not only by the end-effector but also by every part of the manipulator. They are branches (with a certain thickness), wooden poles, iron poles, and metal wires.

Even though they are grouped under the definition of **hard obstacles**, during the annotation process and also during the segmentation of the image, the objects are treated as separate classes in order to help the model identify certain types of

classes and to avoid to mislead the training of a particular class by considering objects of different types.

The **soft obstacles** are defined as those obstacles that are not critical in terms of trajectory planning and the contact of the manipulator with this type of obstacle does not lead to damage or to the failure of the grasping task.

Soft obstacles are, for example, leaves and small/terminal branches. The identification and the segmentation of **soft obstacles** in an image is not easy and it can be hard, for the model, to identify every single leaf. Moreover, the annotation part can be even more difficult and labor-intensive since, in a single image, there can be hundreds of leaf instances.

5.1.1 Dataset Annotation

The dataset annotation is a human-intensive task and it can't be easily automated. Since the target of this project is to define a scene in which the manipulator has to operate in an environment that has many obstacles (branches, poles, metal wires, etc...), it is fundamental to recognize these elements in the observed scene.

Figure 5.1: Fully annotated image, all the defined classes are present in the image

According to this assumption, it has been decided to classify the object in the images into 4 categories as can be seen in figure 5.1:

- **Apples**
- **Branches**: this class takes into consideration also the wooden poles that are used to give a structure to the orchard and as a support to the metal wires.
- **Metal wires**: there are mainly 2 kinds of metal wire. The first one is made of metal and it does not have any covering (rubber or plastic), while the second one is characterized to have a white plastic cover that makes the wire thicker.
- **Background**: the background is characterized to have a high variation of pixels in the associated mask. This is the example of the grass (on the ground) and all the farther trees (even though they have some apples on them). Note that this class has been added to obtain a fully labeled image. In this way, everything that is not classified as apple, branches, metal wire, or background can be considered as **leaves** and so as **soft obstacle**.

As mentioned above, there is another category that would be more complicated to manually annotate in a dataset: the **leaves**.

Annotating the leaves would be time-consuming and counterproductive because one of the main problems that occur in object detection (and specifically in YOLO) is that sometimes if the masks of two objects are too close or one object partially hides another one, the mask results in being merged, meaning that there would be just one instance for two actual objects.
For this application, it would not be a problem, since the important thing is to obtain a **fully labeled point cloud**, so it is not important to define the boundaries of a single instance or object (except for apples, which instances must be precise in terms of the definition of the mask and the number of instances present in the scene).

The provided dataset is composed of **602 images** coming from different orchards and in different settings (both meteorological and lighting conditions) to achieve better accuracy and to avoid some training errors, and in particular **measurement biases**. For this reason, the variation of the dataset composition is not only focused on the different changes of the settings but regards:

- **the apples**: there may be red, green, yellow, and shaded apples
- **the vegetation**: plants usually vary according to the typology of the apple, highlighting a strong difference in the leaves shape and size
- **layout and settings**: the layout of the orchard can be extremely different from farm to farm and it depends on the farmer's choice. In this case, there is also a variation in lighting and weather conditions.

All the images have a fixed resolution of 640x480 pixels and, this resolution will also be the same as the images used for the inference during the detection phase.

The tool that has been used to do this task is **SALT** (Segment Anything Labeling Tool) which is based on the model developed by Meta (**SAM** - Segment Anything Model) [16].

It has been decided to split the entire dataset in the following way: the first 80% of the images has been dedicated to the **training** (480 images), while the remaining 20% has been dedicated to the **validation** (119 images). Furthermore, it has been created an additional set of unseen images in order to visually test the performances of the network and its ability to correctly detect new images.

SALT - Segment Anything Labeling Tool

One of the primary models employed is the **Segment Anything Model** (SAM), developed by Meta [16]. SAM is a robust vision foundation model designed to generalize segmentation across diverse images and objects. By leveraging prompt-based inputs such as points, SAM provides a versatile and highly adaptive segmentation capability. The model was trained on an extensive dataset comprising over 1 billion masks and 11 million images. Its architecture is designed to be generalizable, and capable of segmenting objects across various image domains without retraining or fine-tuning, making it a powerful tool for image analysis tasks in a broad range of applications.

To facilitate dataset creation and annotation for the specific use cases in this study, the **SALT** (Segment Anything Labeling Tool) tool was employed [17]. SALT provides a user-friendly interface to efficiently annotate datasets using the SAM model's segmentation outputs. It enables rapid, interactive labeling by allowing users to refine SAM's initial predictions, correcting segmentation boundaries and adding class labels where necessary. This capability is particularly valuable in scenarios where large-scale annotated datasets are required for model training or evaluation, ensuring that the generated annotations maintain a high degree of accuracy and consistency.

The combination of **SAM**'s automated segmentation and **SALT**'s refinement tools played a pivotal role in creating the annotated datasets used in the experiments, thereby enhancing the overall efficiency and reliability of the data preparation process.

The integration of these tools aligns well with the objectives of this research, as both **SAM** and **SALT** contribute significantly to overcoming challenges related to the manual annotation of large datasets, reducing annotation time, and improving the quality of the final datasets.

This tool was really efficient with the detection of clear and **color dependent objects** like apples and branches among the leaves. This is most likely because of

the fact the **SAM** is extremely dependent on a region-growing algorithm and it works perfectly in the identification of objects that are characterized by the same pattern or color.

In opposition, this model underperforms in the recognition of objects that have a variation of patterns due to some particular lighting conditions or due to some adjacent elements that have similar patterns to the selected object (for example when there are two adjacent apples in a low light environment).

Another point of weakness is that this model tries to select and suggest a mask of the bigger object. This can be a problem during the identification of the **metal wire**. In many pictures, metal wires were located just in front of the fruit, making it difficult, for the model, to create a suitable correct mask. The identification of metal wires is critical for the planning of the trajectory and for the definition of the availability of the fruit itself.

Furthermore, sometimes the metal wire was not visible enough in the image (bad lighting conditions) and it was too small to be detected, particularly with dark backgrounds and/or rusty wires.

5.1.2 Training

The following section regards the training phase and all the training parameters that have been tuned in order to obtain a suitable optimized and efficient model.

Optimizers

This section provides an in-depth discussion of optimization techniques used in deep learning, specifically focusing on three optimizers: **Adam**, **AdamW**, and **Stochastic Gradient Descent (SGD)**.

Adam Optimizer

The **Adam** (Adaptive Moment Estimation) optimizer [18] is a widely used optimization algorithm in deep learning. It computes adaptive **learning rates** for each parameter by maintaining running averages of both the gradients and their second moments.

How Adam Works

Adam maintains two-moment estimates:

- **First moment estimate (mean):** It calculates the exponential moving average of the gradient.

- **Second moment estimate (uncentered variance):** It calculates the exponential moving average of the squared gradient.

The update rule for each parameter is given by:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where:

- α is the learning rate.
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected first and second moment estimates, respectively.
- ϵ is a small constant to avoid division by zero.

Metrics and Performance

Adam adapts learning rates for different parameters based on the first and second moments, leading to faster convergence. It works well with sparse gradients and has been shown to perform well in practice across a wide range of neural network architectures.

AdamW Optimizer

AdamW [19] is a variant of **Adam** introduced to address a common issue known as "*L2 regularization decay.*" In **Adam**, the weight decay (L2 regularization) is included in the gradient update, which can interfere with the adaptive learning rates. **AdamW** decouples weight decay from the gradient updates, leading to better generalization.

How AdamW Works

AdamW uses the same moment estimates as Adam but modifies the update rule as follows:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \alpha \cdot \lambda \cdot \theta_t$$

where:

- λ is the weight decay factor.

The weight decay term is applied separately from the adaptive learning rate updates, ensuring that regularization is consistent across iterations.

Metrics and Performance

AdamW generally leads to better generalization than Adam due to the improved weight decay mechanism. It is particularly effective in scenarios where overfitting is a concern.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) [20] is a fundamental optimization algorithm used in deep learning. Unlike standard gradient descent, which uses the entire dataset to compute gradients, **SGD** updates the model parameters using a single example (or a mini-batch) at a time.

How SGD Works

SGD updates the parameters as follows:

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta L(\theta_t)$$

where:

- α is the learning rate.
- $\nabla_\theta L(\theta_t)$ is the gradient of the loss function with respect to the parameters.

Metrics and Performance

SGD is known for its simplicity and efficiency. However, it is sensitive to the choice of the learning rate and can suffer from issues like slow convergence and getting stuck in local minima. Techniques like momentum and learning rate schedules can be applied to improve its performance.

Comparison of Optimizers

Convergence Rate

Adam and **AdamW** typically converge faster than **SGD**, especially in cases involving sparse gradients. The adaptive learning rates in **Adam** and **AdamW** allow them to make larger strides in regions with low gradients while being cautious in regions with high gradients.

Moreover, **Adam** and **AdamW** tend to be more stable compared to vanilla **SGD**. The moment estimates help *smooth the optimization process*, reducing oscillations and leading to more stable convergence.

Generalization

While **Adam** and **AdamW** converge faster, they sometimes overfit, leading to worse generalization. **AdamW**'s decoupling of weight decay generally provides better generalization than **Adam**. In contrast, **SGD**, particularly when combined with momentum, often generalizes better due to its less aggressive learning dynamics.

Memory and Computation Efficiency

SGD is more memory-efficient as it does not require maintaining additional moment estimates. **Adam** and **AdamW** require additional storage for the first and second moments, which could be a concern in memory-constrained environments.

Applicability

- **Adam:** Preferred for complex models and cases with sparse gradients.
- **AdamW:** Suitable when regularization and generalization are priorities.
- **SGD:** Effective for large-scale datasets and scenarios where computational efficiency is critical.

To summarize this comparison, it is possible to state that **Adam** and **AdamW** provide faster convergence and stability but can struggle with generalization. On the other hand, **SGD** is simple and often leads to better generalization, with the small compromise a slower convergence. The choice of optimizer depends on the specific application and priorities such as speed, stability, or generalization.

5.2 Sensor Matching

The second part of the system is the so-called **Sensor Matching** or the **Sensor Matching Algorithm**. This part regards the implementation and the integration of the **Vision system** with the ROS infrastructure and the **pipeline** that will be explained later. To have a better understanding of how the system is structured, there is the necessity to explain how the **pipeline** works and how the communication between ROS nodes works.

5.2.1 Pipeline

As it can be seen from figure 5.2, the **pipeline** node is the main node and inside it, there is the decision logic that characterizes the system.

The pipeline node is the core of the system and it manages the states of the robots, the detection outputs, the planning nodes, and the grasping clients.

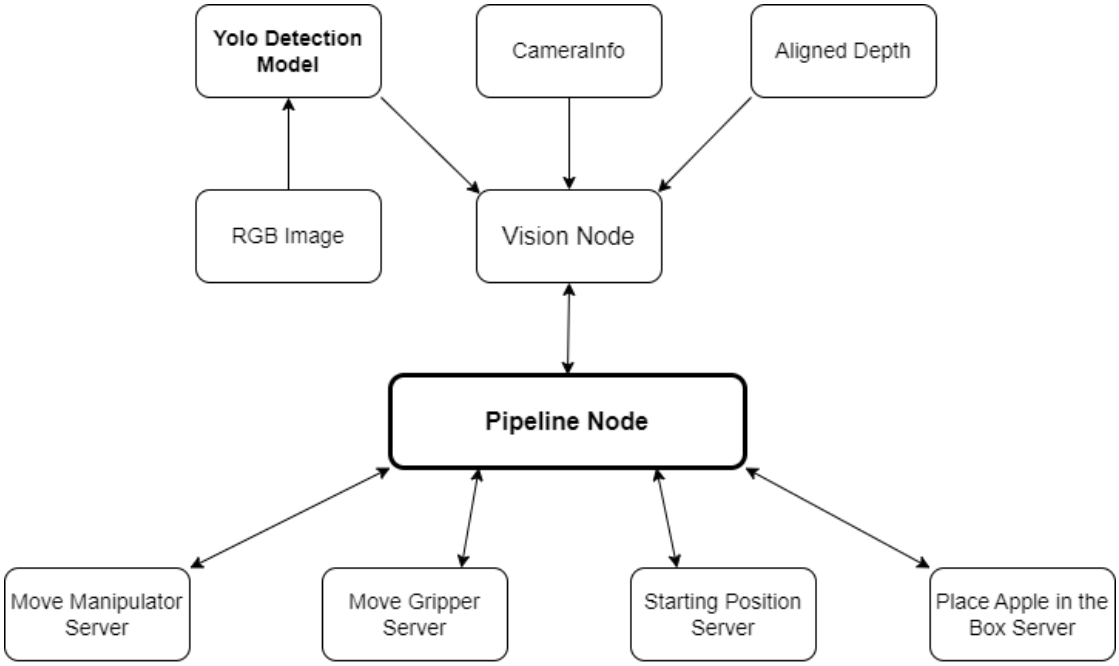

Figure 5.2: Schematic System Description

The **pipeline** is a **state machine** in which there are, not only the control statements that can call different functions of the system, but there are some intrinsic clients and publishers that guarantee the trajectory planning feasibility or the correctness of the detected apple pose.

The pipeline is composed of 7 States and it works as follows:

Algorithm 1 Apple Picking Pipeline Algorithm

```
Initialize system setup variables, state = "SETUP"
while True do
    Check if previous step is complete
    State ← pipeline_state
    if state == "SETUP" then
        Start setup process
        Open gripper (set position to "open")
        Move to "HOME" state
    else if state == "HOME" then
        Move to the starting position
        If successful, set state to "INIT"
    else if state == "INIT" then
        Update the scene (clear octomap and update environment)
        Set state to "GET_APPLE_POSE" if successful
    else if state == "GET_APPLE_POSE" then
        Request apple pose
        Publish collision sphere for the detected apple
        Set state to "PLAN_AND_EXECUTE"
    else if state == "PLAN_AND_EXECUTE" then
        Move manipulator to the apple's position
        Remove collision sphere after successful move
        If successful, set state to "GRASP"
    else if state == "GRASP" then
        Close the gripper to grasp the apple
        Set state to "PLACE"
    else if state == "PLACE" then
        Place the apple in the final position
        Return to "SETUP" state after successful placement
    else
        Handle any failure, retry, or return to the home state
    end if
end while
```

To have a better understanding of the state diagram, figure 5.3 can be taken into account, which may help to identify the state transition conditions and to clarify the working principle of the pipeline.

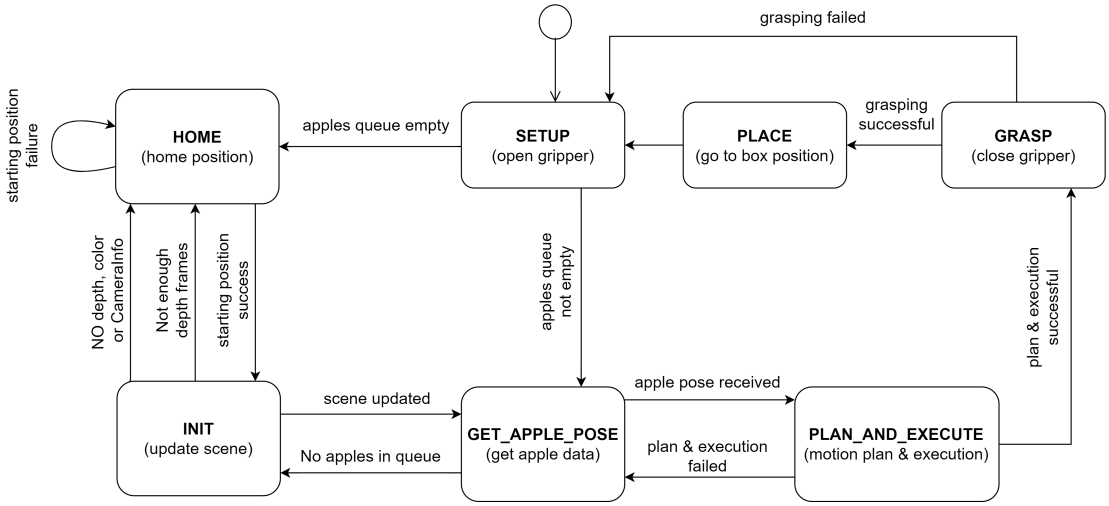

Figure 5.3: State Diagram of the system

5.2.2 Init Node: the perception task

Now that a better overview of the working principle of the pipeline has been provided, it is possible to accurately describe the function of the perception task: the so-called **INIT** node.

The **INIT** node is a central node that manages the visual inputs and it is able to generate the targets and the obstacles. This task is heavily dependent on the YOLOv8 model previously trained on the custom dataset and on the hardware on which the system is running on.

The task and the function that is responsible for what has been described above will be called, from now on, `update_scene`. The manipulator keeps a fixed starting pose for the entire duration of `update_scene` to have a better understanding of the environment and to have a better precision regarding the understanding of the scene. The starting pose has been chosen in order to simplify the objects recognition (usually apples can be viewed more easily from a lower position) and to better optimize the grasping orientation (later explained).

The starting position can be observed in figure 5.4 and it has been obtained by imposing the angular position for each joint. This position is reached, whenever there is not any apple to be picked in the apple's queue, so the `update_scene` can be subsequently called after the manipulator movement toward the target position.

Figure 5.4: Manipulator starting pose

The `update_scene` function can be divided into 4 main blocks:

- **Inference**
- **Mask creation**
- **Points projection**
- **Point cloud segmentation**

In the following paragraphs, there will be a discussion and a detailed description of how each of the 4 functions works.

5.2.3 Inference

The **inference** is a fundamental step in the identification and perception of the scene. The input needed for this task is just the topic regarding the RGB image. The resolution of the image is 640x480 (the same resolution has been used for the images with which the YOLOv8 model has been trained). The advantage of using YOLOv8, for the inference is that the image data is processed only once by the model, so there is no need to have multiple inferences. This results in a faster model (with respect to RCNN and Fast-RCNN) and allows the system to have a competitive cycle time. Even though the system can not be considered a proper real-time application (because the stream of the data is not continuous and there

are not any deadlines of the tasks) the usage of a fast model is necessary to have a minor usage of resources.

The outputs of the inference are multiple and many of them will be used in the following phase: the **msask creation**. For example, for each detected element of the scene, it is available:

- Box position and size
- Classification Confidence
- Contour of the mask of the detected object
- Id of the object
- The label of the object

5.2.4 Mask creation

The mask creation is the next step of the `update_scene` function and it is responsible for the generation of the masks for each detected class. In particular, starting from the first detected object of the list containing all the instances that the model found during the inference, the node is able to create a closed figure from the contour of the selected object.

To better explain all the side features that have been added to this particular function that aims at finding the masks for each class, it must be said that the final outputs of the **msask creation** are:

- a grayscale image that contains the mask of the **branches**
- a grayscale image that contains the mask of the **metal wires**
- a grayscale image that contains the mask of the **background**
- a list of grayscale images in which there are the masks of all the **apples**

Regarding the **branches** and **metal wire** masks, the approach that has been followed to get the masks is really simple: starting from the contours, if the object's confidence is above a determined threshold (tuned experimentally), the mask is added to the final one. During this operation, the final mask is obtained by imposing the value of 1 in the pixels in which the mask is detected and the value 0 in the pixels in which it is not.

This decision has been made in order to provide a clearer overview of the debugging phase and to make the visualization easier. Moreover, the decision to use binary values for the presence of an instance is motivated by the facility in managing points in which 2 masks may have been overlapped.

As regards the **apples**, as previously mentioned, it has been decided not to use a single mask in which to store multiple instances of apples, but a list has been opportunely created. This vector contains the instances of those apples whose confidence is over a given threshold (every class has its own threshold).

Moreover, the apple mask has been shrunk by a few pixels compared to the original one to avoid some problems during the **point projection phase**. The problem is related to the uncertainty that characterizes the projection of tangent points. The problem is that, when a point belongs to a surface, which tangent plane is oriented the same way as the line that connects the camera frame and the point itself, it may happen that the point is erroneously projected to an element that is on the background (behind the actual apple point).

By shrinking the apple mask by few pixels it is possible to avoid that kind of problem but it can be more difficult to estimate the radius of the apple.

For the last mask, the one that concerns the **background**, the logic is similar to the one that brought to the creation of the **branches** and **metal wire** masks. Also in this case, the threshold related to the minimum confidence that a background object must have to be added to the final background mask can be set separately.

The final result of this step, in terms of image, is the following:

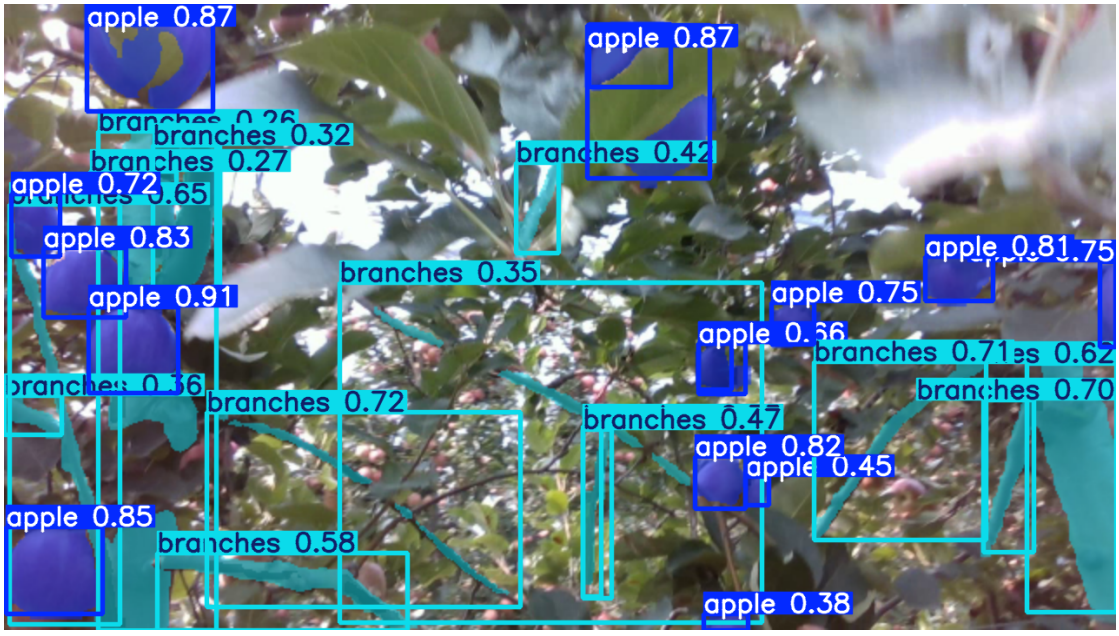

Figure 5.5: Random labeled image

In figure 5.5 it is possible to see the output of the previous step, the inference phase, in which the model performs a detection on a given image and produces the contours of the objects. In figure 5.6, the white items are the apples (each apple

Figure 5.6: Random labeled image in grayscale, with all the classes

has its own mask associated), while the grey elements represent the branches. The padding related to the shrinking of the apple mask, in terms of pixels, has been set to 0 pixels (so it represents the original size of the apple mask).

The mask has been obtained with the following thresholds:

- **apples confidence** = 0.5
- **branches confidence** = 0.33
- **metal_wire confidence** = 0.1
- **background confidence** = 0.1

As shown in figure 5.6, all the elements with confidence above the fixed threshold are added to the scene and can be later projected in the space to create a point cloud.

5.2.5 Points projection

The **points projection** is managed by the function `depth_to_xyz`. The inputs of this function are outputs of the previous step (the **mask creation**): a list that contains the masks of each **apple**, the mask of the **branches**, the mask of the **metal_wire** and the **background**'s mask.

With these masks, the **points projection** will check, one by one, the pixels of every image and project only the points that belong to a given mask. Each point will then be saved in its related list of projected points based on the label. A schematic representation of how the system manages the inputs and the outputs is shown in figure 5.7.

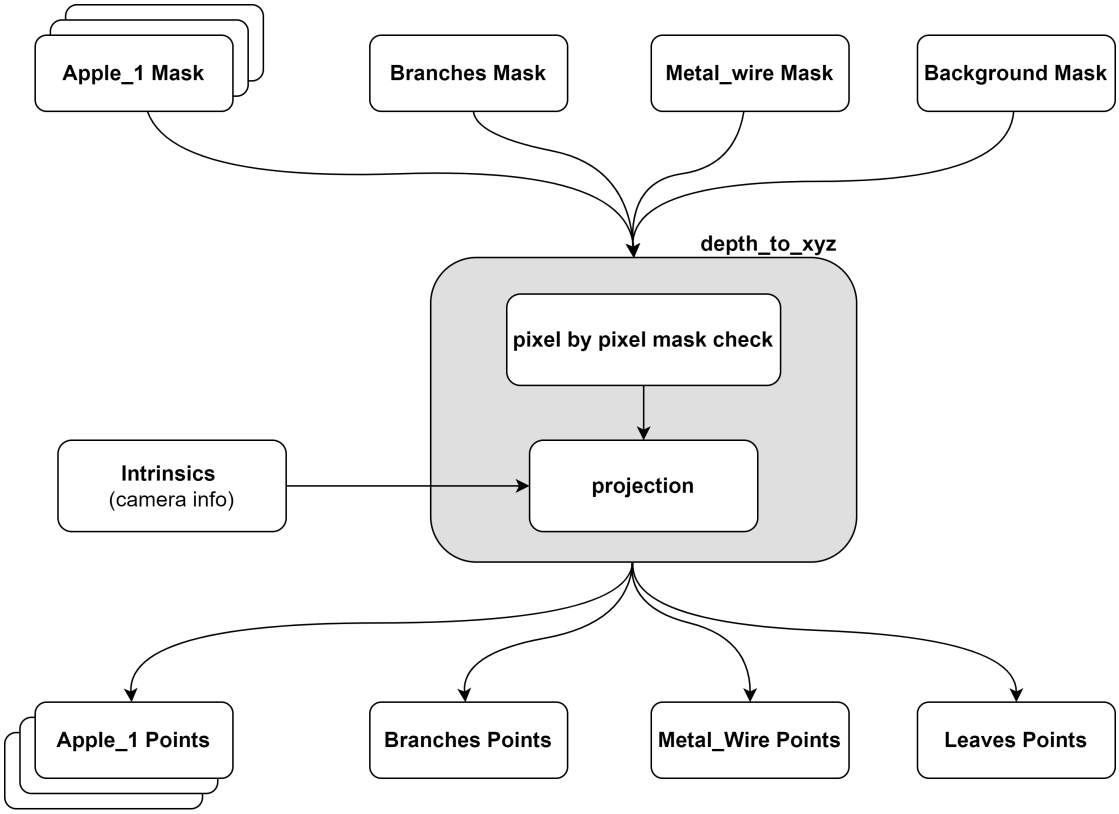

Figure 5.7: Schematic representation of the `depth_to_xyz` function

This function is responsible for the point projection according to the following theory.

The calculation of 3D coordinates of a point in the camera frame given an RGB image and depth information uses the intrinsic camera parameters, which include focal length and principal point (c_x, c_y), also called *centroid*. For each pixel in the RGB image, its 2D coordinates (u, v) and the depth value $d(u, v)$, corresponding to the distance from the camera to the point, are needed. The 3D coordinates in the camera's frame (X, Y, Z) can be computed as:

$$X = (u - c_x) \times \frac{d(u, v)}{f_x}$$

$$Y = (v - c_y) \times \frac{d(u,v)}{f_y}$$

$$Z = d(u,v)$$

Here, f_x and f_y are the focal lengths along the x and y axes, respectively. The depth value $d(u,v)$ provides the Z coordinate, while the X and Y coordinates are derived by projecting the pixel location into 3D space using the camera intrinsics and the depth value.

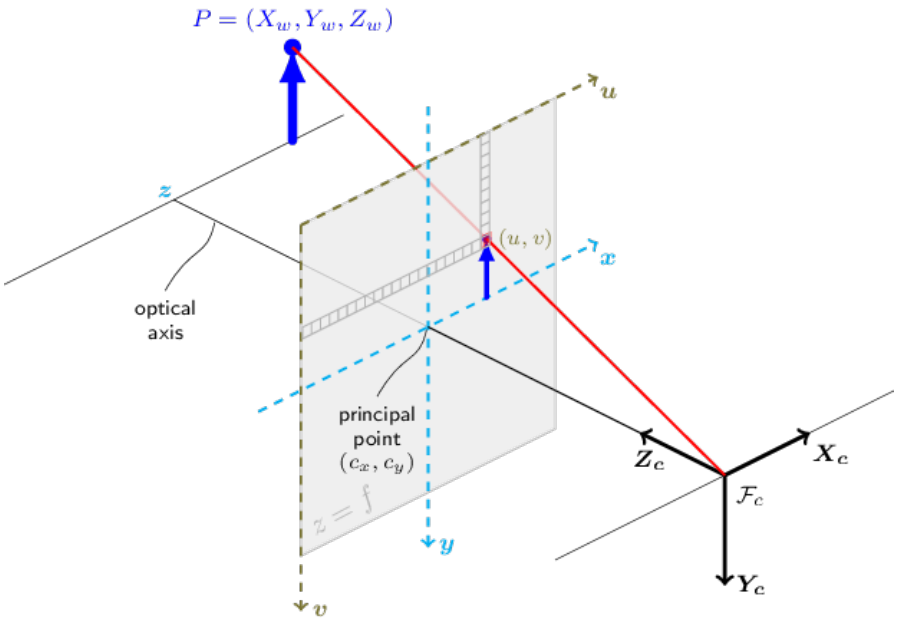

Figure 5.8: Point projection from image and camera frame [21]

The outputs of the **points projection** phase are the following:

- a list that contains the points of the **branches**
- a list that contains the points of the **metal wires**
- a list that contains the points of the **leaves**
- a list of list in which there are the points of each **apples**. Each list corresponds to a determined apple

The list of points related to the leaves is obtained to the projection of those pixels that are not classified in any of the categories previously mentioned (so if the pixel is neither labeled as an apple, nor branch, nor metal wire nor background).

This approach has some drawbacks linked to the fact that there may be the possibility to have some unseen and unlabeled items on the scene, that are not classified in any of the classes defined in the model training, and subsequently, they may be defined as leaves.

5.2.6 Point cloud segmentation

The **point cloud segmentation** is the last step of the `depth_to_xyz` and it provides the outputs that must be present in the manipulator scene in order to plan a suitable trajectory, trying to avoid collisions, with the goal of reaching a target point (apple).

The inputs of this step are the ones coming from the **points projection** step and they are lists of points (lists contain the coordinates of the points in an ordered fashion, starting from the x to the z of a point and, then starting again with the x of another point).

The points related to each apple are managed in a different way with respect to the ones that belong to the classes that define the obstacles (branches, leaves, and metal wires). The *obstacle classes* are given to a particular function that is able to generate a point cloud message (*PointCloud2* in ROS2), in which, the corresponding reference frame has been set equal to the one in which the points are evaluated (`camera_color_optical_frame`).

To better understand the deployment of the points in the space, a suitable color code was chosen for the different classes.

- branches → yellow
- leaves → green
- metal wire → blue

The apple points receive a different treatment, that is designed to create a virtual representation of an apple in the planning scene. This virtual representation and its utility will be explained later in the section related to collision avoidance.

For every list of **apple points** will follow the steps reported below:

- **Center calculation**: all the points belonging to the same apple are averaged so that it can be possible to have the center of the apple. Since the size of the resulting mask is not the same as it should be (See section 5.2.4), the projected points cover less than half of the actual apple surface. This means that the calculated center does not offer an accurate "depth" with respect to the camera frame. In fact, the resulting center appears to be placed right behind the surface of the apple that faces the camera (as shown in figure 5.9

in which there is a representation of what is described). For this reason, there is the need to adjust (experimentally) the "depth" position of the center of the apple.

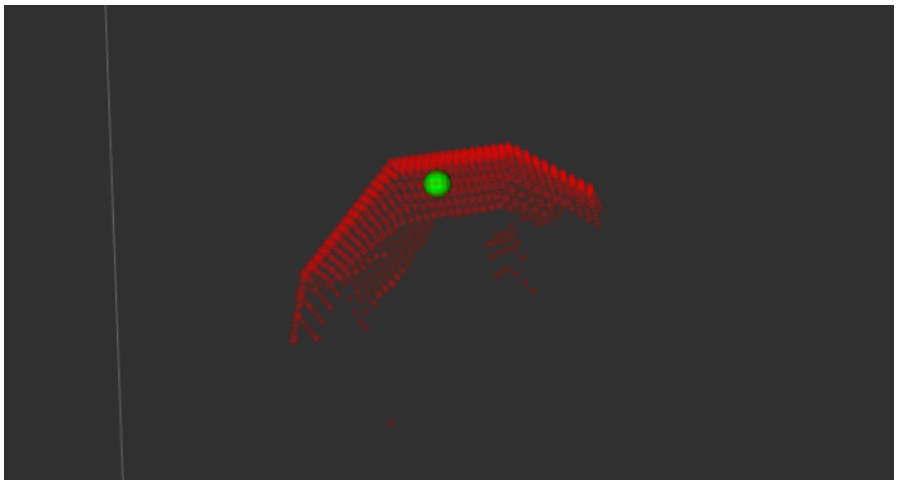

Figure 5.9: Apple center (green) and apple points on the surface (red)

- **Transform of center coordinates**: the coordinates of the center calculated in the previous step are in the frame `camera_color_optical_frame` and they need to be transformed in the `world` frame. This transformation is managed by the function **transform_point** in which it is possible to specify the frame in which the coordinates of the point are available, the point's coordinates, and the frame to which the coordinates need to be transformed.

  The function heavily depends on the ROS2 tf2 package, which manages to record and transform all the frame positions and orientations. The transformation of this point will be needed in the following steps, particularly when a grasping orientation will be calculated.

- **Calculation of grasping orientation**: the calculation of the grasping orientation is a fundamental step of the grasping task. It determines whether an apple can be reached and whether it is possible to have a successful grasping.

  During the execution of this function and for the entire duration of the **INIT** node, the manipulator is fixed in the starting pose shown in figure 5.4.

  The python function responsible for the calculation of the grasping orientation is called `calculate_grasping_orientation` and takes two positions expressed in the `world` frame. The first is the position of the `tool_frame` of the starting pose, while the second one is the position of the center of the apple. The `tool_frame` is positioned between the gripper's fingers as seen in figure 5.10.

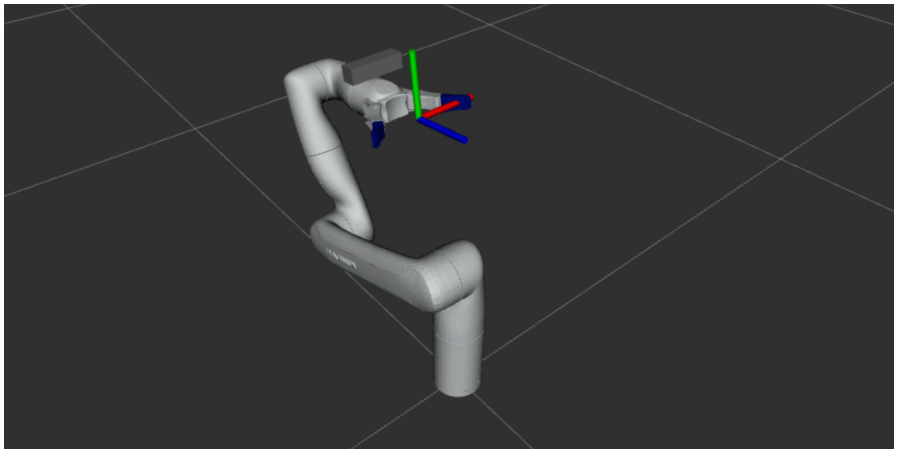

Figure 5.10: Representation of `tool_frame` position

The objective of this function is to return a quaternion that can be used as a representation of a rotation with respect to a reference frame. Another objective is to find the direction of the line that passes through the two previously mentioned points. That direction will coincide with the z-axis of a frame placed in the apple center. Direction with the reference frame can be easily calculated with the following formula:

$$\vec{v} = \frac{\vec{ap} - \vec{tf}}{\|\vec{ap} - \vec{tf}\|}$$

where $\vec{v}$ is the direction of the line that passes through the points, $\vec{ap}$ is the position of the apple, and $\vec{tf}$ is the position of the origin of the `tool_frame`. The next step is to identify the two elements that characterize a quaternion: the **vector** (the direction of rotation) and the **angle** (rotation angle from the reference frame along the direction of the vector).

The direction (different from $\vec{v}$) is calculated in the following way:

$$\vec{dir} = [0,\ 0,\ 1] \times \vec{v}$$

where, the vector $[0,\ 0,\ 1]$ represents the z direction of the world frame (is coincident with the axis of the base link of the manipulator).

The angle has been easily calculated in this way:

$$\alpha = cos^{-1}([0,\ 0,\ 1] \cdot \vec{v})$$

The quaternion is finally defined as:

$$Quat = [cos(\alpha/2),\ sin(\alpha/2)\ \vec{dir} \cdot \hat{i},\ sin(\alpha/2)\ \vec{dir} \cdot \hat{k},\ sin(\alpha/2)\ \vec{dir} \cdot \hat{k}]$$

- **Additional frame rotations**: the final position of the resulting frame must be further rotated to guarantee that the frame rotation on the apple in the correct one. The correct orientation of the apple frame is determined by the fact that the planner (later explained in the 5.3) will try to create an overlapping between the two frames, in such a way that each direction will coincide.

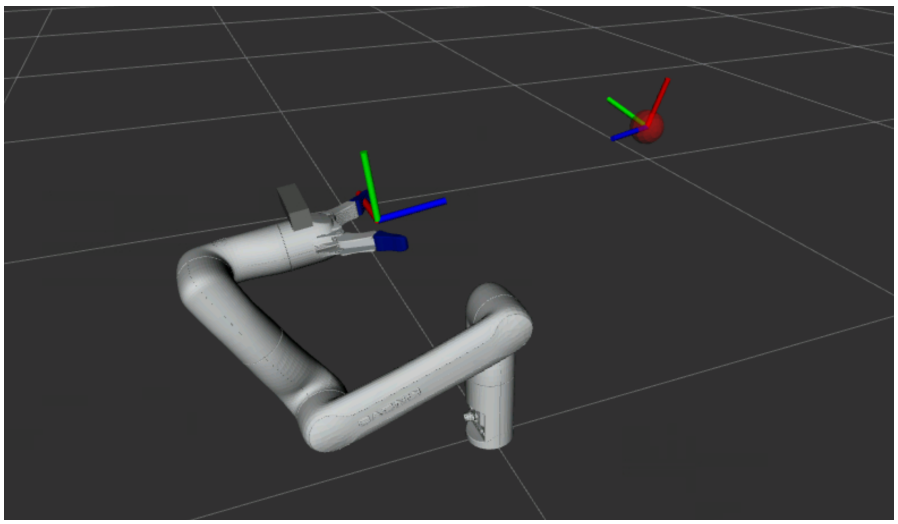

Figure 5.11: Apple frame before additional rotations

As seen in figure 5.11, to obtain a suitable and correct frame rotation, the current `apple_frame` must be rotated by 180° around y (green) and later rotated by 90° around z. Since the rotations are subsequent and related to the current frame (not the original reference frame, the `world` frame), the quaternion *Quat* must be right-multiplied in the following way:

$$Quat_{final} = Quat \cdot [0,\ 0,\ 1,\ 0] \cdot [\frac{\sqrt{2}}{2},\ 0,\ 0,\ \frac{\sqrt{2}}{2}]$$

The final result is displayed in figure 5.12.

- **Apple center shift**: as described in the step **Center calculation** and depicted in figure 5.9, the estimated center of the apple does not exactly represent the center, because it is obtained by averaging the coordinates of the points of an apple. A solution may be the experimental shift of the apple center. This shift has been designed to be along the direction that connects the origin of the `tool_frame` and the previously calculated apple center as seen in figure 5.13. The value of this shift can be easily changed since it is the parameter of the `shift_point` function in Python.

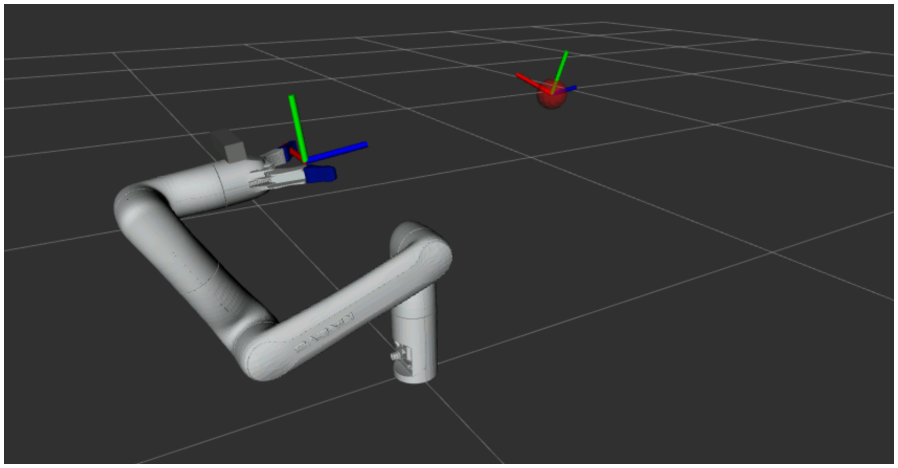

Figure 5.12: Apple frame after additional rotations

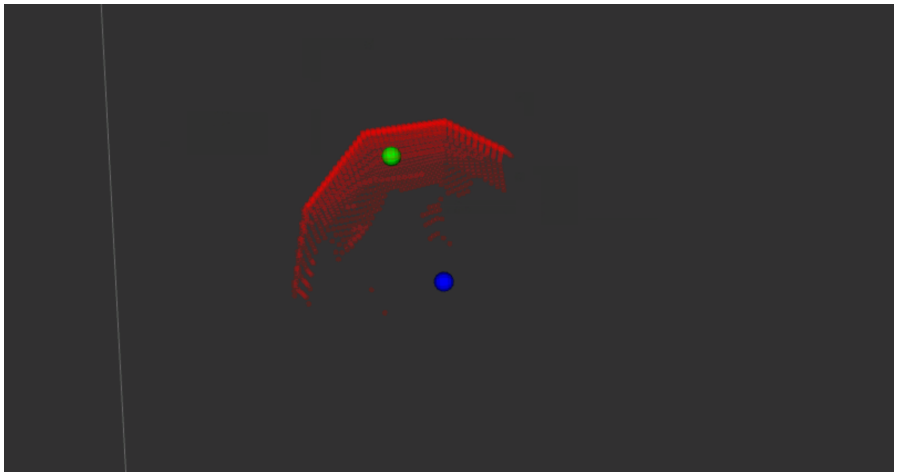

Figure 5.13: Shifted apple center (blue) and previously calculated center (green)

- **Radius Estimate**: the previous step, the shifting of the estimated apple center, is a fundamental step for the definition of apple data. In fact, in order to have complete and sufficient information about detected apples, the radius information might be necessary for the definition of the planning scene. For example, during the first planning, starting from the starting pose, the manipulator must have a clear understanding of the defined obstacles, and it must take into account also the detected apples since a possible trajectory planning could cause the collision of an apple, that can result in a little movement and lead to a failure of the "picking process".

  For this reason, there is the need to create a virtual sphere that can replace

and simulate the position and size of the apples in the scene.

The apple radius has been calculated by simply averaging the first 20% of the distances between the shifted apple center and all the points that were projected for a given apple. It has been decided to average just 20% of the distances because there may be some outliers that can lead to an overestimated radius. Apple radius above a certain threshold brings a failure in the trajectory planning because the fully opened gripper could get in contact with the surface of a big collision sphere.

Moreover, it has been added a collision cylinder positioned just above the apple (shown in figure 5.14) to avoid that the planned trajectory could make the manipulator collide with the petiole of the apple, causing a possible oscillation or the detachment of the apple itself.

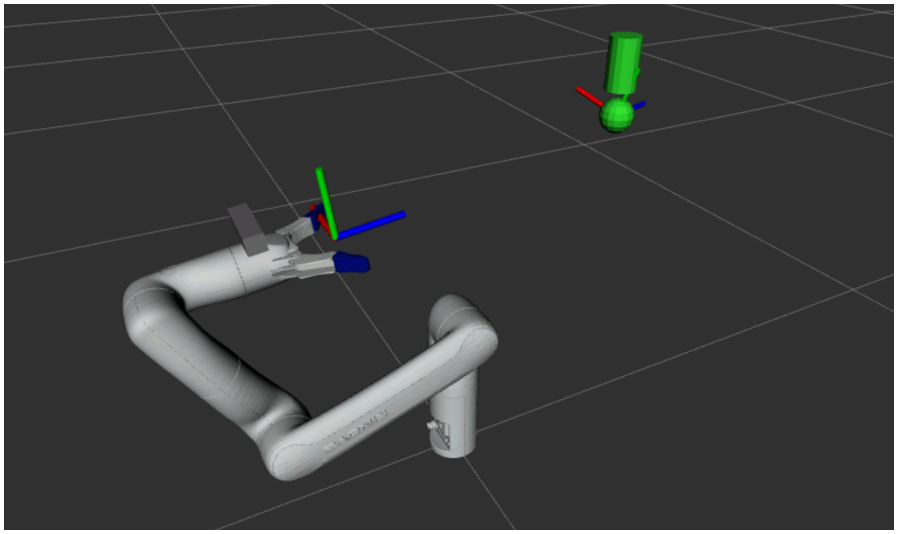

Figure 5.14: Collision objects generated for each apple

- **Addition to the queue**: it concludes the process that manages the information about the apple points and it simply creates a Python dictionary variable for every object with the following information:

```
{
    "apple_id": self.apple_count,    # counter for detected apple
    "center": {
        "x": shifted_appl_cntr[0],
        "y": shifted_appl_cntr[1],
        "z": shifted_appl_cntr[2],
    },
    "orientation": orientation,      # quaternion type
    "radius": radius,
    "distance_to_camera": distance_to_camera,
}
```

5.2.7 Get_Apple_Pose Service

An important service called **get_apple_pose** has been implemented in order to guarantee efficient communication between the **pipeline** node and the **vision** node. The server manages to empty the apples queue by first sending back (to the client that, in this case, is the pipeline node) the information about a single apple. In particular, the server manages to send a custom service type that has been properly defined. The service type is the following and it has been created in the ROS2 package named **apple_pose_srv**.

```
---
geometry_msgs/PoseStamped pose
std_msgs/Float64 radius
bool success
string message
```

As it can be seen from the service definition, the client (pipeline node) sends an empty request and receives the pose of the apple, the radius, the success state, and a possible message. Usually, the message can be the number of apples left in the queue, so that, the manipulator can optimize the cycle time by avoiding performing another detection or wasting time to go to the starting pose. The case in which the queue is not empty can be clearly understood through figure 5.3.

Moreover, if no apple is in the queue or no apple has been detected in the scene, the server sets the success parameter to `False` and the message to the following statement: *"No apples in the queue"*.

5.3 Trajectory Planning, Obstacle Avoidance and Grasping

The last section of the system description is dedicated to trajectory planning and everything that regards the movement of the manipulator in the environment. As mentioned in section 5.2, the definition of the planning scene is essential for the successful accomplishment of the motion planning task since, unseen objects or wrongly projected points can lead to the impossibility of the definition of a suitable trajectory or the collision with possible hard obstacles. The platform used to manage path planning, obstacle avoidance, and trajectory execution is **MoveIt 2**.

MoveIt 2 works on top of `ROS2` and provides interfaces to multiple motion design libraries, enabling design paths that guide the robotic manipulator safely from origin to target (apple) and ensure that obstacles are avoided.

This section examines in detail the planners used in this work, focusing on the technical aspects of each algorithm, and the design of collision avoidance.

5.3.1 Overview of MoveIt 2

MoveIt 2 is a powerful motion planning tool designed for robotic applications. It allows us to integrate environmental sensing data with trajectory generation, ensuring that the manipulator can adapt to changes in the surroundings. A key feature of MoveIt 2 is its ability to handle high-dimensional configuration spaces, which are typical in robotic manipulators due to the multiple degrees of freedom (DOF) involved.

The planning process begins with an evaluation of the robot's current state in the planning scene, which includes the robot model, environmental objects, and the target (apple). This scene is continuously updated using point cloud data, and the goal is to compute a path that avoids collisions with obstacles such as branches, leaves, and metal wires, while guiding the manipulator towards the apple. Unfortunately, there is no possibility of having a dynamic update of the planning scene because the system collects the inputs needed to reconstruct the hard obstacles point cloud from the RGB image, which requires the inference through the YOLOv8 model. The inference speed is limited by the hardware and, since a greater accuracy (in terms of confidence of detected objects) is required, the time needed to achieve a satisfactory result is not compatible with the real-time goal of scene updating.

5.3.2 Motion Planners in MoveIt 2

MoveIt 2 provides access to several motion planning algorithms, each suited to different types of environments and problem constraints. The planners selected for

this project are based on their suitability for real-time pathfinding in environments filled with obstacles and narrow passages. The following sections will delve into the technical details of these planners and their algorithms.

5.3.3 Rapidly-exploring Random Trees (RRT)

The **Rapidly-exploring Random Trees (RRT)** algorithm is a sampling-based planner that incrementally builds a tree by sampling random points in the configuration space and attempting to connect them to the existing tree. The strength of RRT lies in its ability to efficiently explore large, high-dimensional spaces, which makes it ideal for manipulators with many degrees of freedom.[22]

In RRT, the search starts from the robot's initial state and proceeds by generating random configurations. Each configuration is checked for collisions, and if it is valid, the planner attempts to expand the tree towards it. This process continues until the planner either finds a valid path to the goal (the apple in our case) or the search times out. RRT is particularly effective in environments with narrow passages, such as the gaps between branches.

The underlying mechanics of RRT involve a distance metric that guides the expansion of the tree. At each iteration, the planner samples a random point in the workspace and identifies the nearest vertex in the tree. The planner then attempts to move from that vertex toward the sample using a local planner. If the movement is collision-free, the sample is added to the tree. Otherwise, a new random sample is selected. This allows the algorithm to efficiently handle complex environments without explicitly discretizing the space. [23]

Advantages of RRT

- Efficient exploration of large spaces with high degrees of freedom.
- Well-suited for finding paths in environments with narrow passageways.
- Scales well with complex robotic arms, making it ideal for multi-DOF manipulators.

Limitations of RRT

- The paths generated by RRT are often suboptimal, leading to jagged and indirect routes.
- The randomness inherent in the algorithm can result in inconsistent planning times.
- Post-processing is often required to smooth the paths.

5.3.4 RRT-Connect

RRT-Connect is an enhanced version of the standard RRT algorithm. It builds two trees instead of one: one tree grows from the robot's current position, and the other grows from the goal (apple). The key idea behind RRT-Connect is to quickly find a connection between the two trees, thereby speeding up the search process [24].

This bidirectional search is highly recommended when there are no direct paths from start to goal due to obstacles. In the case of the apple-harvesting task, the manipulator often has to navigate around multiple obstacles like branches and wires. By expanding both from the start and the goal, RRT-Connect significantly reduces the time it takes to find a feasible path.

Technically, RRT-Connect alternates between extending the start tree and the goal tree, checking for possible connections between the two trees after each expansion. The result is a faster convergence to a solution, even in environments where obstacles densely populate the workspace [24].

Advantages of RRT-Connect

- Faster than RRT due to bidirectional search.
- Handles complex, cluttered environments efficiently.
- Produces fewer unnecessary movements compared to standard RRT.

Limitations of RRT-Connect

- Still retains some of the suboptimality issues of RRT, meaning paths may need further optimization.
- Its performance can degrade when the goal is in a highly constrained region of space.

5.3.5 Probabilistic Roadmap (PRM)

The **Probabilistic Roadmap (PRM)** planner is another sampling-based method, but unlike RRT, it is a multi-query planner. PRM works by first randomly sampling points in the free space to generate a roadmap. This roadmap consists of *milestones*, which represent valid configurations, and *edges*, which represent feasible paths between those configurations [25].

PRM is particularly effective in static environments, where the roadmap can be precomputed and reused for multiple queries.

The construction phase of PRM involves two steps:

1. **Node Generation**: Random points are sampled in the configuration space, and each sample is checked for collisions. Valid samples are added as nodes to the roadmap.
2. **Roadmap Construction**: Pairs of nodes are connected if they can be joined by a collision-free path, typically determined using a local planner.

Advantages of PRM

- Efficient for solving multiple queries in static environments.
- The roadmap can be reused across different planning problems in the same environment.
- Generates smooth paths when obstacles remain unchanged.

Limitations of PRM

- Inefficient in dynamic environments, as the roadmap must be frequently updated.
- Performance degrades when obstacles are moving, as in our apple-harvesting scenario.
- High computational cost when constructing the roadmap for high-dimensional spaces.

5.3.6 Optimization-Based Planners

While sampling-based planners like RRT and PRM are well-suited for finding feasible paths in complex environments, the paths they generate often require refinement. MoveIt 2 includes optimization-based planners like **CHOMP** and **STOMP**, which aim to smooth and optimize the trajectories after an initial path is found.

5.3.7 CHOMP (Covariant Hamiltonian Optimization for Motion Planning)

CHOMP is an optimization-based planner that improves the trajectory by minimizing a cost function that incorporates both smoothness and collision avoidance [26]. Once a rough path is generated by a sampling-based planner like RRT-Connect, CHOMP can refine the path by adjusting it in small increments to reduce sharp turns and ensure that the manipulator moves fluidly through the workspace.

CHOMP operates by iteratively adjusting the trajectory based on the gradient of the cost function. This gradient is computed by evaluating the robot's proximity to obstacles and the smoothness of the path. CHOMP excels in environments where smooth, continuous motions are critical, such as when navigating around densely packed obstacles like branches.

Advantages of CHOMP

- Produces smooth, collision-free trajectories.
- Can handle highly constrained environments.
- Efficient for path refinement once a rough plan is available.

Limitations of CHOMP

- Requires a good initial guess to start optimization.
- Performance is sensitive to the tuning of cost function weights.
- May converge to local minima, resulting in suboptimal solutions.

5.3.8 STOMP (Stochastic Trajectory Optimization for Motion Planning)

STOMP is another trajectory optimization method, but unlike CHOMP, it uses stochastic sampling to explore multiple trajectories simultaneously [27]. STOMP is particularly effective in noisy or cluttered environments, where randomness can help escape local minima that deterministic methods like CHOMP might fall into.

STOMP works by generating multiple candidate trajectories around the initial path and evaluating them based on a cost function. The planner then uses the best trajectories to update the path iteratively, gradually improving the overall quality.

Advantages of STOMP

- Robust to noisy and cluttered environments.
- Capable of escaping local minima, producing globally optimal trajectories.
- Useful for refining paths produced by other planners.

Limitations of STOMP

- Performance depends heavily on the quality of the initial path.
- Computationally expensive due to its stochastic nature, so it implies that it is more time expensive.
- May require many iterations to converge, especially in highly complex environments.

5.3.9 PointCloudOccupancyMapUpdater

The `PointCloudOccupancyMapUpdater` is a crucial plugin in MoveIt 2 for managing dynamic collision checking by incorporating real-time sensor data into the planning scene. This plugin processes incoming point cloud data and updates an internal occupancy map. The occupancy map is a voxel-based 3D grid where each voxel represents a portion of space that may either be free, occupied, or unknown.

The workflow begins by subscribing to a point cloud topic (in this case the point clouds are published by the sensor matching module).

The `PointCloudOccupancyMapUpdater` inserts the points into an octree structure, which is used to dynamically maintain the state of the 3D space. Each voxel within the octree can be updated to reflect whether it is free (no obstacles), occupied (obstacle present), or unknown.

A key feature of this plugin is its ability to filter out noise and unnecessary data points. This is done using parameters such as `max_range`, which limits the sensor data to a specific distance, and `voxel_resolution`, which controls the granularity of the occupancy map. Additionally, it can ignore points below the robot's base to avoid collisions with the ground or platform. The updated occupancy map is then fed directly into the MoveIt 2 planning pipeline, where it is used during path generation and collision checking. This enables the robot to dynamically adapt to changes in the environment, such as moving obstacles or newly introduced objects, making it particularly useful in real-world, cluttered environments like orchards, where branches and leaves are continuously moving.

Chapter 6

Test and Results

6.1 Training Results

The very first attempts of training (manually tuned) showed mediocre results for what concerns apples and branches, and background. Sometimes, the mask of the background was slightly bigger than it should have been, meaning that possible branches or metal wires (hard obstacles) can be confused and they can be regarded as non-harming. This decision can have serious implications for the scene around the manipulator, but most importantly on the planning of the trajectory.

To better optimize the choice of the hyperparameters it has been decided to use a framework called **Optuna** [28]. Optuna is a performant optimization tool that allows the refinement of the hyper-parameter that can be chosen to tune a specific model.

6.1.1 Optuna settings

Optuna is based on the Tree-structured Parzen Estimator (TPE), which uses SMBO (Sequential Model Based Optimization) to build a model based on the set of hyper-parameters and it tries to optimize (maximize or minimize) a defined variable based on previous models. In this specific case, the most relevant variable to be optimized is the $mAP50-95_{average}$, which is a result that can be extracted at the end of each training and represent the mean average precision of the instances that have an IoU (Intersection over Union) that is in between 50% and 95%. In particular, for this project, it has been decided to use the average of the $mAP50-95$ of all the classes (apples, branches, metal wires and background).
The settings regarding the Optuna optimization are the following and they regard mainly the optimization of the hyper-parameters linked to the dataset augmentation:

- **epochs**: range between 100 and 300. It represents the number of epochs in

which the optimizer tries to change the model weights in order to minimize the loss function.

- **optimizer**: [AdamW, SGD]. As commented in the previous chapter, this are the optimizers that are being chosen to train a suitable model.
- **batch**: 8 (to avoid out-of-memory problems). It represents the number of elements (training images) that are taken into consideration in one iteration of the training.
- **box**: range between 0.02 and 4.0. It represents the weight that has been assigned to the box loss in the loss function to be minimize. In particular, has to take into account how well is the guessed box positioned with respect to the truth box.
- **cls**: range between 0.02 and 4.0. It represents, similarly to the previous case, it represents the weight of the term of the loss function that is related to the classification loss.
- **hsv_h**: range between 0.0 and 1.0. It represents the value associated with the hue of the image. A hue change is particularly useful in training a model that can maintain good performance in different lighting conditions.
- **hsv_s**: range between 0.0 and 1.0. It is the parameter linked to the saturation. The correct modification of this parameter can affect the ability of the model to recognize the object in different environmental conditions successfully.
- **hsv_v**: range between 0.0 and 1.0. It represents the brightness of an image. As for the **hsv_h**, the correct tuning of this parameter can help the model to be performant also in different lighting conditions.
- **degrees**: range between -180 and 180: This parameter is responsible for the random rotation of an image within the dataset. The rotation is particularly useful when it is needed to identify objects in different orientations.
- **translate**: range between 0.0 and 1.0. It represents the translation of the image. This translation results in the image being shifted by a fraction of its size and it is useful to recognize objects that are partially hidden.
- **scale**: range between 0.0 and 0.9. It represents the scaling factor to be applied to an image so that a model can identify objects that have a different size or are away from the camera.
- **shear**: range between -180 and 180. It represents the shear effect that is applied to an image. This effect is effective to simulate an object being viewed from a different angle.

- **perspective**: range between 0.0 and 0.001. This feature allow a perspective transformation in such a way that the model can perceive the 3D properties of an object.
- **flipud**: range between 0.0 and 1.0. This value is connected to the probability that a given image is flipped upside down. This affects the model's ability to detect objects in different positions in the image.
- **fliplr**: range between 0.0 and 1.0. Similarly to the **flipud**, this value represents the probability to have an image flipped from left to right.
- **mosaic**: range between 0.0 and 1.0. This hyper-parameter is linked to the possibility to create "mosaic style" image, obtained by the conjunction of cropped 4 images.
- **mixup**: range between 0.0 and 1.0. It is linked to the possibility to blend two different images (together with their labels), to improve the ability of the model to generalize by introducing some noise.
- **copy_paste**: range between 0.0 and 1.0. This parameter is particularly useful to increase the number of total instances. It copies and pastes an object of an image into another one.

6.1.2 Optuna results

The first 50 trainings showed the following result in terms of which hyper-parameter is more important, and how much it can influence the the trend of the $mAP_{average}$. As it can be seen from figure 6.1, *the Optimizer* is the hyper-parameter that makes the result change the most. For this reason, to avoid wasting time in training and to have a clearer overview of what hyper-parameter (linked to data augmentation) can have the biggest impact on the final result, the optimizer has been kept fixed for the next trainings (by manually imposing one of the 2 optimizers among AdamW and SGD).

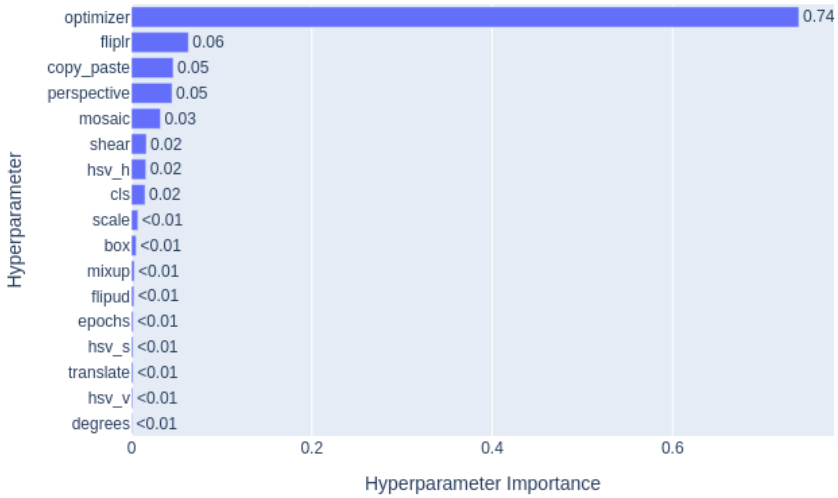

Figure 6.1: Importance of hyperparameters for the training

It has been necessary to perform 172 trainings (110 with AdamW and 62 with SGD) to have the possibility to analyze the data and to collect a sufficient amount of metrics in order to make a suitable optimal choice about the model to be used for detection.

As it can be seen in figure 6.2 the trend of the $mAP_{average}$ for AdamW trainings, is not linear and it does not improve visibly as the hyper-parameters are updated. This can suggest the difficulty of the optimizer to improve the overall result even though the possibility to change a modest quantity of hyper-parameters. In opposition, the performances and the trend of SGD trainings are good in terms of improvement, meaning that the optimizer can correctly pursue and find a local minima.

Furthermore, after locking the "*optimizer*" parameter that can opportunely be changed from Optuna settings, the graph reporting the importance scores for each optimization parameter changed as well, as shown in figure 6.3.

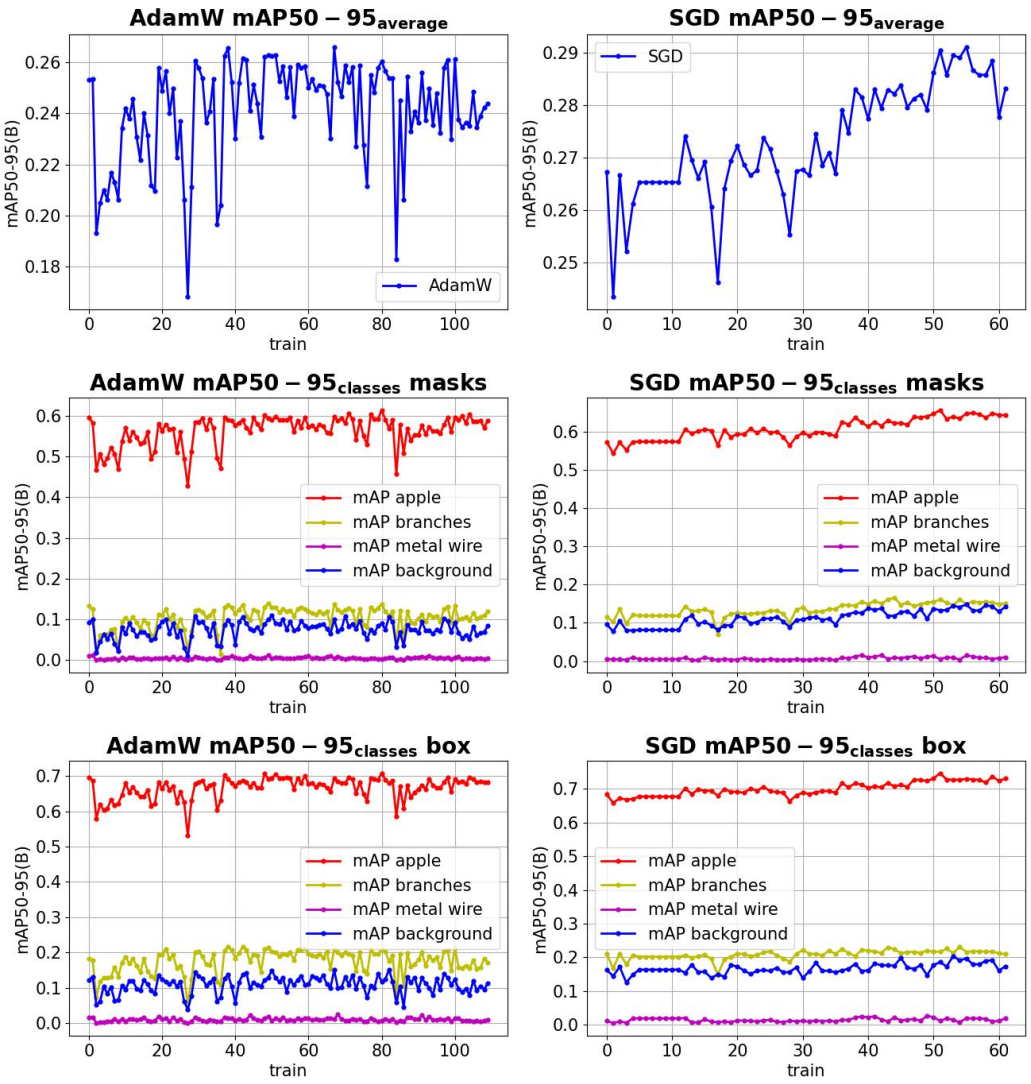

Figure 6.2: mAP50-95 (Average and by class) for AdamW and SGD trainings

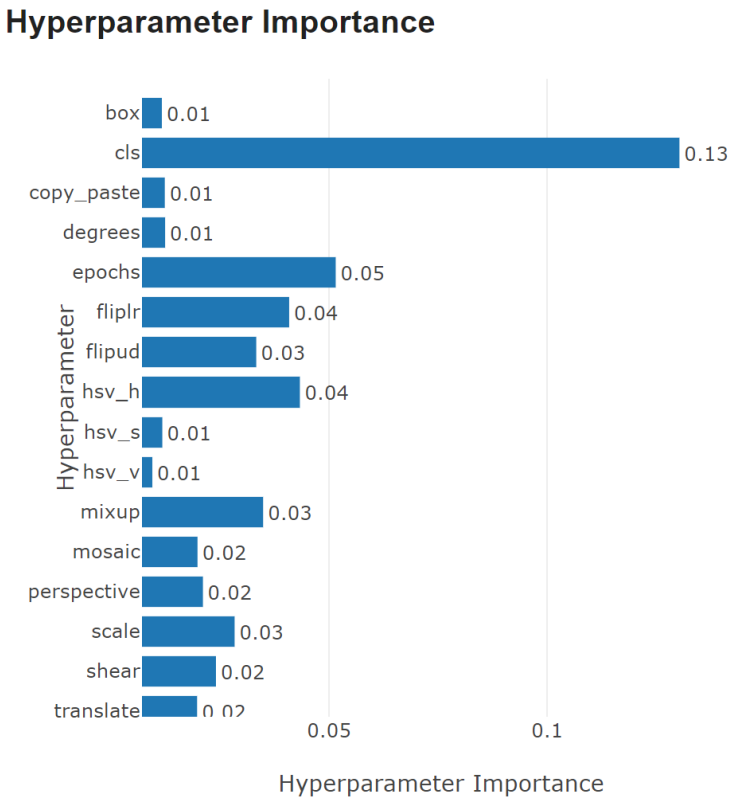

Figure 6.3: Importance of hyper-parameters for training without considering the choice of the optmizer

The plot below represents the trend of the value of the objective function to be optimized by Optuna. In this case, since the minimizer was set to "MINIMIZE" mode, the quantity to be minimized resulted to be $(1 - mAP50 - 95(B))$.

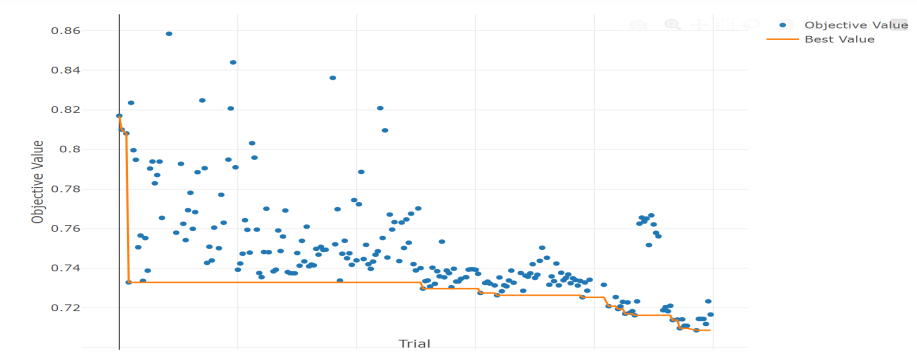

Figure 6.4: Optimization history and objective function values

Model training results and considerations

The poor results of the hyper-parameter optimization can be brought back to the composition of the dataset. The dataset has different kinds of images coming from different scenarios and orchards but there may be some intrinsic and unwanted characteristics that can influence the final performance of the model: the biases.

The biases are of different types:

- **Selection Bias**: this bias is such that some labels have few samples or the data quality is very poor. The common reason for the selection bias is that some data are difficult to collect.
- **Measurement Bias**: this error is correlated with the fact that sometimes, the measurement tools used to create the training dataset are different with respect to the validation dataset's instruments and setting.
- **Confirmation Bias**: this bias deals with the intrinsic biases that shape everyday human lives. These kinds of biases are difficult to detect because the majority think that the world has the same vision, implying there may be some social aspect to be considered.

In this specific case, especially during the tests that were performed in the laboratory, the most significant sources of errors are the selection bias and the measurement bias.

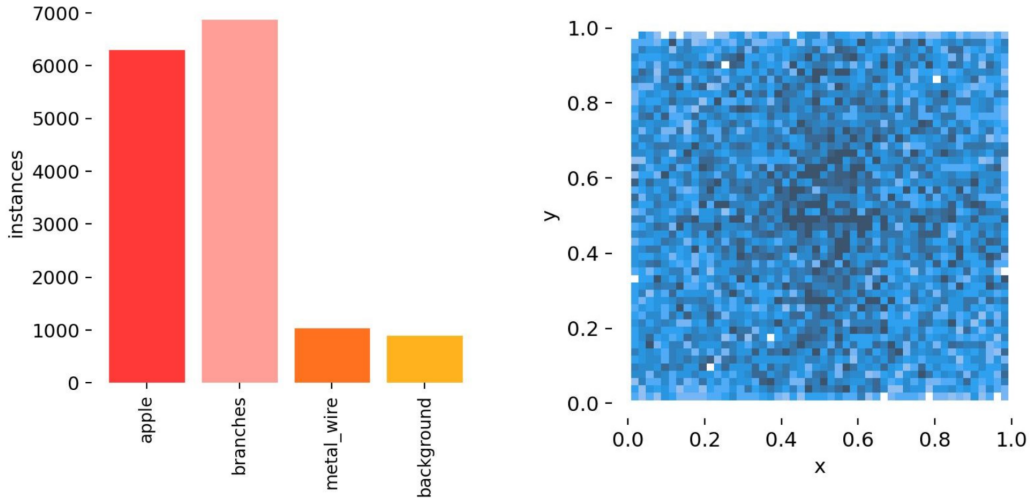

Figure 6.5: Number of instances in the dataset

The measurement bias is highly present for the fact that, the starting YOLOv8 model (the model from which the final network has been trained) was pre-trained

for the apple class, so the performances related to that class were clearly better than the other classes. Moreover, the total number of instances is unbalanced (see figure 6.5) and can heavily influence the result even though a data augmentation process has been implemented in YOLO's trainer.

For what regards the positions of the elements (related to the centers of the bounding boxes), there is a good sparsity in the data and it is optimal in terms of the expected generalization capabilities of the model.

The overall most performant model is the best model of the training called `train_257`, which has an objective function value of 0.708 ($mAP50-95_{average}$ of 29.2%), has been obtained through the following hyper-parameters setting (SGD Optimizer):

box	0.364	**fliplr**	0.210
cls	0.267	**flipud**	0.028
copy_paste	0.166	**hsv_h**	0.136
degrees	59.873°	**hsv_s**	0.473
mixup	0.294	**hsv_v**	0.998
mosaic	0.515	**epochs**	256
perspective	0.001	**shear**	-16.453°
scale	0.225	**translate**	0.877

Table 6.1: Final optimization hyper-parameters best model overall (SGD)

The metrics of the model obtained with the parameters above are summarized in the table 6.2

mAP50-95(B)	29.2%	**Optimizer**	SGD
box mAP apple	73.0%	**mask mAP apple**	64.7%
box mAP branch	21.6%	**mask mAP branch**	14.9%
box mAP metal_wire	2.0%	**mask mAP metal_wire**	1.5%
box mAP background	19.7%	**mask mAP background**	14.8%

Table 6.2: Performance metrics of model `train_257`

Moreover, in order to evaluate the performance of the model, it has been selected the best model obtained using the AdamW optimizer, which corresponds to the training called `train_100`, that has been obtained with the following settings:

box	0.198	**fliplr**	0.093
cls	0.273	**flipud**	0.340
copy_paste	0.783	**hsv_h**	0.051
degrees	37.752°	**hsv_s**	0.217
mixup	0.245	**hsv_v**	0.380
mosaic	0.720	**epochs**	203
perspective	0.0	**shear**	1.583°
scale	0.473	**translate**	0.017

Table 6.3: Final optimization hyper-parameters best model (AdamW)

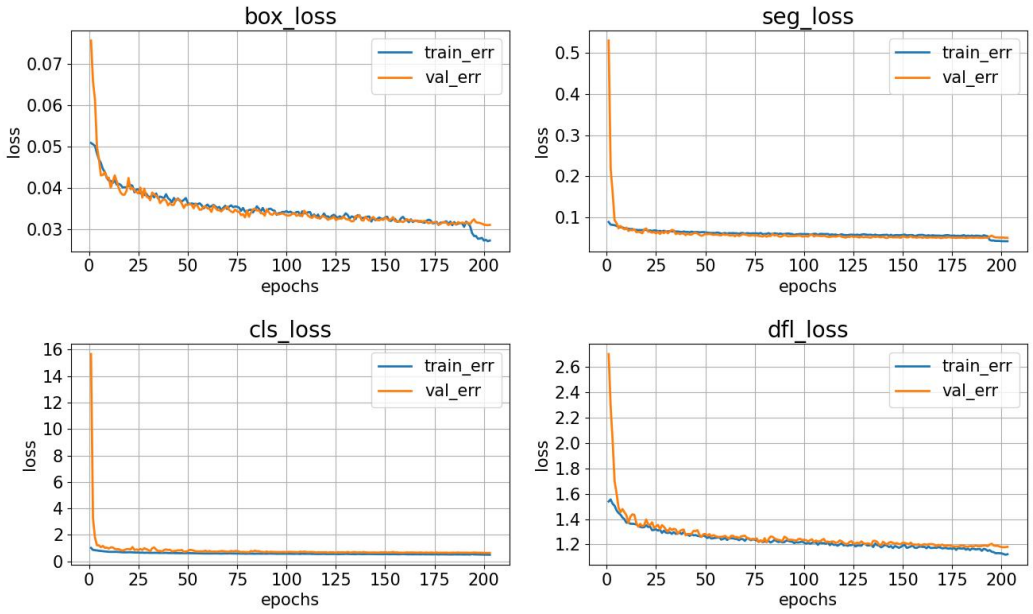

Figure 6.6: Validation and training losses during training

The trend of the losses (both validation and training) are reported in figure 6.6. As can be seen in the trend of the *box_loss* and in the *cls_loss* (classification loss), there is a slight sign of overfitting during the final epochs of the training (validation and training error diverges). Even though the last model cannot be satisfactory and reliable, the internal logic of YOLO's trainer can memorize the best model (saving its weights).

Moreover, in figure 6.7 it is possible to have a clear idea of the trend of the different losses and of the mAP50-95 for the entire training duration. The trend of some metrics, like precision and recall, shows a high variance growth since some

unstable classes are sensible to a slight weight change in the network architecture. These classes are the branches and the metal wires, especially considering the instances that have small masks.

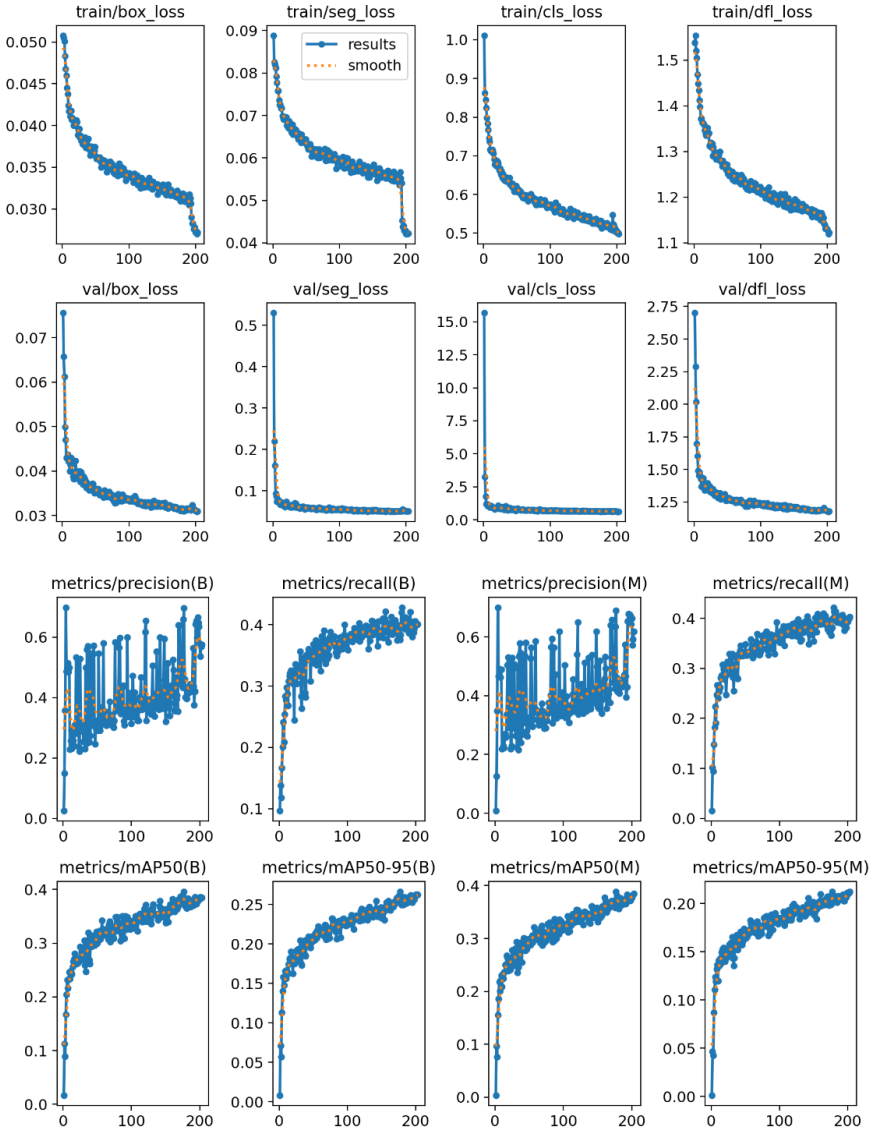

Figure 6.7: Metrics and training variables for model `train_100`

The specific metrics related to this model are reported in the table below (table 6.4):

mAP50-95(B)	26.6%	**Optimizer**	AdamW
box mAP apple	69.3%	**mask mAP apple**	59.7%
box mAP branch	20.9%	**mask mAP branch**	13.6%
box mAP metal_wire	1.1%	**mask mAP metal_wire**	0.4%
box mAP background	15.1%	**mask mAP background**	10.0%

Table 6.4: Performance metrics of model `train_100`

6.2 System Testing

The testing of the system and the working capabilities of every single function and part that intrinsically works within the system involves the design of a particular test set, the objective of which is to collect useful metrics.

The metrics can be the means through which it is possible to compare the system with the actual state-of-the-art. In particular, the collected metrics are useful to have a clear idea about the performance of the system for the following capabilities:

- **Apple pose estimation**: the sensor matching algorithm's ability to easily estimate an apple's position (center).
- **Performance of the segmentation model** (partially verified, system tested in a single environment).
- **Inference time**: time needed to perform an inference operation in the machine on which the system is running on
- **Cycle time**: time needed for the manipulator to get an apple and place it in the collection box.
- **Repeatability**: the ability of the system to repeat actions and get the same result in the same working conditions (i.e., in the apple position estimation).

6.2.1 Setup

The tests were conducted in the PIC4SeR laboratory, where it was possible to create a simulated environment (see figure 6.8) with real plants (chestnut branches and leaves).

The used robotic arm (described in section 4.3), has been placed and fixed at the edge of the desk, while plants and other floating branches were placed starting from the other side of the desk, near the wall. Some papers (representing the background

of the stage) were placed on the wall's surface to create a more realistic environment that would help the segmentation model to avoid incurring on measurement biases.

Figure 6.8: Test setup (manipulator and environment)

ROS2 nodes and the Python scripts were running in a Intel NUC 11 PAH, powered by an 11^{th} gen. Intel@ Core i5-1135G7 and as Mesa intel@ Xe Graphics a graphic processor. The machine was running Ubuntu 22.04.4 LTS and ROS2 Humble.

During the tests, since the default MoveIt2 planner (RRT-Connect) does not provide the possibility to add multiple point cloud sources by giving different "importance" or weights for the trajectory planning optimizer, it has been decided to not consider the class of the leaves. In this way, only branches and metal wires are added to the planning scene and later transformed into octomap (by the `PointCloudOccupancyMapUpdater`).

6.2.2 Tests results

The test scenarios are a type of cluster of test sets in which, the goal, the setting, and the software configuration remain the same. During each test scenario, multiple attempts are performed and evaluated by the system by means of a dedicated code that stores the data in apposite files. The data collected from the system were then analyzed to create and produce the performance metrics.

The two first tests are meant to verify and compare the different candidate models that were selected in section 6.1.2. After the evaluation of these tests, a final model has been selected for the remaining test sets.

Figure 6.9: Adamw model (`train_100`) detection

As it can be seen in figure 6.9 and 6.10, the elements of the model `train_100`, trained with AdamW, show a better result in terms of confidence score for every class. This result can be brought back to the fact that AdamW is more robust and the final result is slightly dependent from the hyper-parameter tuning. In opposition, model `train_257`, trained with SGD optimizer, requires a deeper understanding

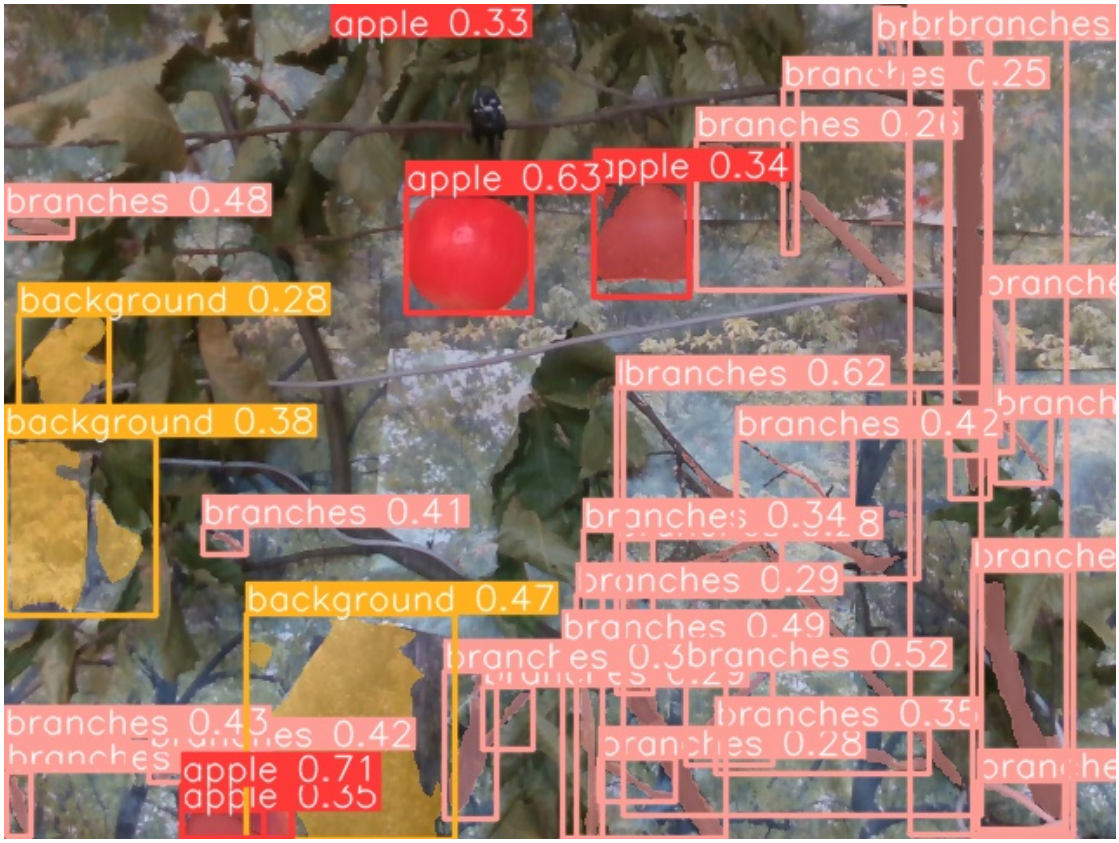

Figure 6.10: SGD model (`train_257`) detection

of all the hyper-parameters (not only the ones linked to data augmentation but also to the intrinsic parameters of the solver and the optimizer itself).

The remaining tests were conducted with the best model, the one called `train_100` and whose specifications and performances are reported in the table 6.4.

	Mean x $[m]$	**Mean y** $[m]$	**Mean z** $[m]$	**Std x** $[m]$	**Std y** $[m]$	**Std z** $[m]$	**Mean Motion time** $[s]$	**Std M. time** $[s]$
1	0.3907	-0.0317	0.6613	0.0095	0.0579	0.0333	11.8561	4.3530
2	0.3907	-0.0838	0.6834	0.0034	0.0076	0.0014	9.1541	1.3943
3.1	0.3855	-0.1273	0.6615	0.0092	0.0116	0.0038	9.5344	1.9845
3.2	0.3701	0.1052	0.5801	0.0154	0.0131	0.0252	8.5007	2.7617
4.1	0.4653	-0.0268	0.4371	0.0129	0.0095	0.0046	10.2826	2.8247
4.2	0.3714	-0.0860	0.6490	0.0048	0.0010	0.0074	8.8779	1.3133
5.1	0.2245	-0.1555	0.6747	0.0056	0.0007	0.0030	7.9517	0.5016
5.2	0.5087	0.2877	0.3611	0.0160	0.0370	0.0109	9.6103	3.5127
6	0.3656	0.1162	0.6040	0.0074	0.0369	0.0111	8.7094	1.5870
7	0.4889	-0.0332	0.3517	0.0464	0.0130	0.0325	11.7390	3.7962
8.1	0.3484	0.1396	0.5771	0.0060	0.0024	0.0141	11.6491	3.7589
8.2	0.4744	-0.0098	0.3687	0.0343	0.0260	0.0386	12.4859	4.8709
9.1	0.3333	-0.0681	0.5977	0.0458	0.0545	0.0123	8.8620	1.5508
9.2	0.4746	0.2218	0.4989	0.0051	0.0030	0.0053	9.6721	2.0673
10.1	0.1622	-0.0289	0.5605	0.0191	0.0158	0.0499	10.4325	4.3200
10.2	0.3949	0.3228	0.7292	0.0065	0.0028	0.0087	10.7747	1.5492
10.3	0.5080	-0.0937	0.4667	0.0023	0.0105	0.0068	9.1003	1.9521

Table 6.5: Apple position accuracy

The table 6.5 represents the metrics related to the apple's positions. In particular, the apples used in each test set are listed in the first column. The first number represents the test number, while the second (if present) represents the apple reference number in that specific test case. As it can be seen from the table, the standard deviation in every direction of the estimated apple pose is really low, meaning that the system is able to provide a repetitive and accurate measurement of a detected object in the scene.

In addition to this result, another consideration needs to be raised. This consideration concerns the fact that, during the repetition of the various attempts within a certain test set, the apple has been manually placed in its original position, so a human error component must also be taken into account. Unfortunately, there was no possibility of measuring this error during the tests, so the resulting standard deviation includes both a human and a system component.

Moreover, it must be clear that the center of the apple is estimated with the application of a shift with respect to the previously calculated position (see figure 5.13) and that this implies that the outliers or the wrongly projected points are not considered and they do not take part in the pose evaluation and radius estimation. The negative aspect of this implementation is that it can badly influence the radius estimation since it heavily depends on the amount of shift that has been experimentally configured.

Another important metric that is going to be discussed is the motion time. The definition of motion time includes the time that the planner takes to create a suitable trajectory, the execution of the trajectory towards the target apple, the grasping, the planning of the trajectory toward the apple collector box, and the execution of the trajectory. The definition of motion time is what can be considered a *cycle time*, which represents the time needed for the manipulator to execute the motion pipeline. The manipulator joint speed has been limited to 80% of the maximum manipulator speed. Moreover, the planner used for all the tests (RRT-Connect, see section 5.3.4) provides a sub-optimal solution in terms of the speed of the joint's rotation, meaning that not always produces neither the fastest path nor the shortest one.

Table 6.6 summarizes the relevant metrics that has been possible to reconstruct from the collected data.

Position estimation			**Motion time**		
Mean $\mathbf{std}_x$ $[cm]$	**Mean** $\mathbf{std}_y$ $[cm]$	**Mean** $\mathbf{std}_z$ $[cm]$	**Average** $[s]$	**1st Quart.** $[s]$	**3rd Quart.** $[s]$
1.4694	1.7841	1.5809	9.9832	7.8607	11.4904
Std in every direction $[cm]$			**Std** $[s]$		
2.8003			2.8259		

Inference time	**Apple grasping rate**	**Collision & Success**
Average $[s]$	**Lower Limit** $[apples/min.]$	**Collision Rate**[1]
6.8382	3.5669	8.91% [29.70%]
Std $[s]$	**Upper Limit** $[apples/min.]$	**Success Rate**
1.1210	5.3252	79.21%

Table 6.6: Result summary table

The resulting cycle time is about 9.9 s, which is comparable to the one obtained by Monash's system [1], remembering that this cycle time do not take into consideration the time needed for the inference.

A parameter that involves the inference time is **Apple Grasping Rate**. In particular, the lower limit has been calculated taking into account that an apple can be picked just after an inference, so that each picked apple corresponds to an image being inferred through the model. In opposition, the upper limit of the **Apple Grasping Rate** has been calculated taking into account that the system has the ability to memorize the positions of all the detected apples and it can execute multiple "pickings" with a single inference.

The discussion related to the **Inference Time** is strictly related to the hardware on which the system is running. More powerful hardware can bring an improvement both in the quality of the image and the inference time.

The **collision Rate** is around 8.91%. The collision situations and cases can be clustered into the following categories:

- **Out of field objects**: Even though the RealSense D435i (section 4.4.1) offers a great compromise between quality, built-in functions, and vision capabilities, in some specific cases, some obstacles may be hidden and outside the field of view of the camera. This condition can be solved by using a different approach for the scene update. For example, it may be possible to realize a "scan" of

[1]It do not considers the total failures plus the collisions, whether they are soft or hard. 29.70% referees to a cumulative failure percentage that involves all the failures that have been registered plus the attempt that collided but succeeded

the environment with multiple orientations of the end-effector in the space, to obtain a comprehensive understanding of the surrounding scene.

- **Weak detection**: The model can be upgraded in terms of reliability and robustness in the recognition of the elements of the scene. Another solution could be to lower the constraints of minimum confidence for each class (especially for branches and metal wires), but this option can have a bad influence on the so-called false positives, which are the elements are assigned another class different from the actual one. This concept creates the risk of neglecting some obstacles that can be potential sources of damage for the manipulator.
- **Scene variation after update**: This problem is related to the time that elapses from the moment in which the RGB frame is captured to be further inferred to the moment the manipulator starts moving after the path planning. During this time the environment can change due to some external factors (i.e. wind, animals, or unexpected events) and this can imply that some objects (hard obstacles in the worst case) can change their position, causing a collision with the manipulator.
- **Objects resolution and dimension**: Elements like metal wires (especially raw ones, without any colored rubber coating) and the branches' terminations, are difficult to be semantically distinguished in a complex environment like an orchard. Moreover, the imprecision of the depth camera and its resolution do not allow a reliable point projection. If some missing points are not present in the planning scene, as in the previously mentioned cases, the manipulator may be led to a collision with these hard obstacles.

Moreover, in some particular configurations, when the branches are too close to the target fruit and the orientation of the frame placed on the apple (that would serve as a reference frame for the trajectory planning), there may be a high probability that the planning could fail.

The image showing the situation described above is depicted in figure 6.11. In that image, the apple and the frame orientation (z pointing toward the wall), together with the voxels, do not allow a feasible grasping, also considering that the manipulator, in an ideal position on the target (frames overlapped), must have a greater amount of free space in the region of the gripper since it approaches with open fingers.

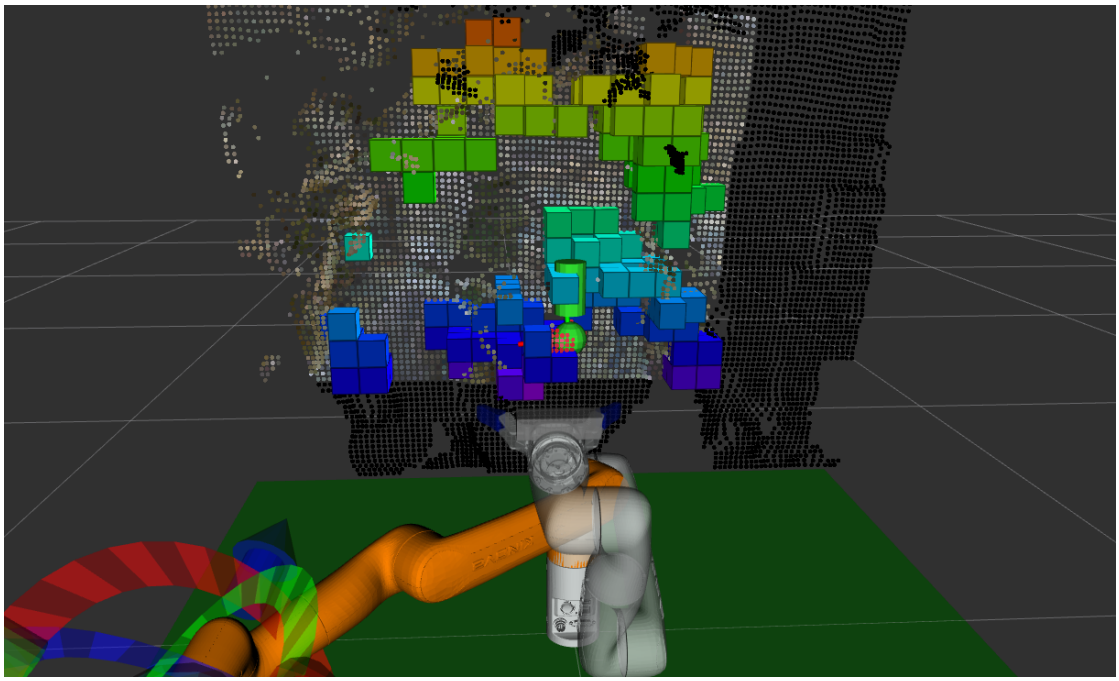

Figure 6.11: Octomap density impeding a feasible grasping

Chapter 7

Conclusion and Future Developement

The project developed in this thesis has been designed to create an autonomous and reliable apple harvesting system that is able to work in not optimized conditions (orchard not carefully pruned or prepared for automatic harvesting). The described system includes a vision module that exploits the power of a fast CNN model like YOLOv8, which has been suitably trained on a custom dataset with the goal of obtaining a fully labeled image. Additionally, the sensor matching module has been designed to match the fully labeled RGB image (coming from the Intel@ RealSense D435i) with the information coming from the depth sensor and the intrinsic of the camera. This module has been essential to reconstructing a fully labeled point cloud and in the determination of the scene obstacles as well as of the target fruits. The pipeline structure played a fundamental role in the management of the activation of the different functions and in the definition of the behaviour of the system with the occurrences of different failure modes. Lastly, the motion planning node, thanks to MoveIt 2 capabilities and planners, is entitled to safely create trajectories for the Kinova Gen 3 Lite manipulator and manage its movement towards the target fruit, trying to avoid the so-called hard-obstacles (branches and metal wires).

The tests that were conducted in the PIC4SeR laboratory brought to following conclusions in terms of metrics and achievements:

- The YOLOv8 model has been successfully trained to guarantee the best achievable performance starting from a custom training dataset. With the help of Optuna [28], it has been possible to reach a satisfactory level of detection in terms of precision and mask conformity.

- The definition of an accurate pipeline made of a Finite State Machine in which,

the presence of 7 states guarantees an effective implementation of precise and accurate task management, whose ultimate goal is to safely guide the manipulator towards the detected targets while avoiding the obstacles.

- The motion and planning system, thanks to MoveIt 2, guarantees a competitive cycle time of 9.983 ± 2.896 s and a grasping rate between 3.567 and 5.325 $apples/min$. The deployment and identification of the object in the space employing the sensor matching module allowed to obtain a competitive success rate (amount of successfully grasped apples among the total) of 79.21% and a collision rate of 8.91%.

7.1 Future Developments

To continuously improve the treated project and make it fully operational and reliable, there may be the need to develop and update some functional parts of the architecture that limit the performance and reliability:

- **Improvement of dataset resources**: An integration of the further images, especially with more instances of the classes that have a lower number of objects (background and metal wires), can reduce the effect of the selection bias and lead to reach a better accuracy in the detection and the understanding of the scene. Furthermore, for what concerns the detection of metal wires, a simple and effective solution could be painting of the external surface of the coating of the metal wire in a color that is distinguishable from the other natural components (blue or white). In this way, the model would make less effort to semantically segment the object in the image. Finally, a greater resolution can be set in the camera configuration to better project the points related to small objects (metal wires or terminations of the branches),
- **Custom planner**: The custom planner could be useful for two separate reasons:
  - reducing the cycle time by optimizing the planning of the trajectory to minimize the motion time.
  - integrating the leaves points in the planning scene (the points obtained in the sensor matching module were not added to the planning scene, see section 6.2.1 for details). The custom planner should be able to give different weights to different objects based on the fact that are categorized as soft or hard obstacles.
- **Apple pose and Radius Estimation refinement**: The improvement in the quality of the estimation of the apple radius could be a high step forward

also toward a more precise pose estimation. For example, having the initial projected points from the RGB image, a clustering algorithm can eliminate the outliers. Moreover, by solving an optimization problem (it could be set in the form of a linear regression), it is possible to have a more accurate estimation of both the radius and center of the apple. The correct estimation of the radius can allow the setting of a dynamic and adaptive closed gripper position. The adaptiveness leads to limited damage to the fruits since it takes care of not trying to squeeze the fingers into a "too close" position.

- **Hybrid planning and dynamic scene update**: The described system is not equipped with a feedback action during the trajectory execution. In a natural environment, the scene can change very fastly and also the target fruit, as well as the obstacles, can vary their position due to some external factors or due to the collision of the manipulator with some connected objects (connected branches, metal wires or unseen connected objects). This limits the system's reliability since it does not respond correctly to fast environmental changes. The usage of MoveIt 2 Hybrid planner [14], it's a powerful instrument that performs complex path planning (using global and local planners) and can adapt quickly to changes in the environment. To have a better understanding of the environment, there could be the necessity of having more performant hardware (with a dedicated powerful GPU) to make more inferences during the manipulator's movement and obtain a feedback action with a constantly updated planning scene.

In the QR code below it is possible to find a demo of the system made during the tests held at PIC4SeR.

Figure 7.1: https://youtu.be/N_dbRIuuCA4

Bibliography

[1] Wesley Au, Hugh Zhou, Tianhao Liu, Eugene Kok, Xing Wang, Michael Wang, and Chao Chen. «The Monash Apple Retrieving System: A review on system intelligence and apple harvesting performance». In: *Computers and Electronics in Agriculture* 213 (2023), p. 108164. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2023.108164. URL: https://www.sciencedirect.com/science/article/pii/S0168169923005525 (cit. on pp. 2, 77).

[2] Long He and James Schupp. «Sensing and Automation in Pruning of Apple Trees: A Review». In: *Agronomy* 8.10 (2018). ISSN: 2073-4395. DOI: 10.3390/agronomy8100211. URL: https://www.mdpi.com/2073-4395/8/10/211 (cit. on p. 4).

[3] Feng Xiao, Haibin Wang, Yueqin Xu, and Ruiqing Zhang. «Fruit Detection and Recognition Based on Deep Learning for Automatic Harvesting: An Overview and Review». In: *Agronomy* 13 (June 2023), p. 1625. DOI: 10.3390/agronomy13061625 (cit. on p. 6).

[4] Jan Martens, Timothy Blut, and Jörg Blankenbach. «Cross domain matching for semantic point cloud segmentation based on image segmentation and geometric reasoning». In: *Advanced Engineering Informatics* 57 (2023), p. 102076. ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2023.102076. URL: https://www.sciencedirect.com/science/article/pii/S1474034623002045 (cit. on p. 8).

[5] Olufunke Vincent and Olusegun Folorunso. «A Descriptive Algorithm for Sobel Image Edge Detection». In: Jan. 2009. DOI: 10.28945/3351 (cit. on p. 11).

[6] John Canny. «A Computational Approach to Edge Detection». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851 (cit. on p. 11).

[7] upGrad. *Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network*. URL: https://www.upgrad.com/blog/basic-cnn-architecture/ (cit. on p. 12).

[8] Evan Weiss, Xin Weng, Tushar Pandey, and et al. «Machine Learning for Computer Vision». In: *MDPI Sensors* 21.4 (2021), pp. 1–25. DOI: 10.3390/s21030874 (cit. on p. 14).

[9] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You only look once: Unified, real-time object detection». In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 779–788 (cit. on pp. 14, 23).

[10] 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». In: *CoRR* abs/1911.11929 (2019). arXiv: 1911.11929. URL: http://arxiv.org/abs/1911.11929 (cit. on p. 16).

[11] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. «Path Aggregation Network for Instance Segmentation». In: *CoRR* abs/1803.01534 (2018). arXiv: 1803.01534. URL: http://arxiv.org/abs/1803.01534 (cit. on p. 17).

[12] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 20).

[13] Kinova. *Together in Robotics*. 2019. URL: https://www.kinovarobotics.com/ (cit. on p. 25).

[14] David Coleman, Ioan A. Șucan, Sachin Chitta, and Nikolaus Correll. «Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study». In: (2014). DOI: http://dx.doi.org/10.6092/JOSER_2014_05_01_p3 (cit. on pp. 26, 82).

[15] Leonid Keselman, John Iselin Woodfill, Anders Grunnet-Jepsen, and Achintya Bhowmik. «Intel RealSense Stereoscopic Depth Cameras». In: *CoRR* abs/1705.05548 (2017). arXiv: 1705.05548. URL: http://arxiv.org/abs/1705.05548 (cit. on p. 28).

[16] Meta. *Segment Anything Model (SAM): a new AI model from Meta AI that can "cut out" any object, in any image, with a single click*. 2019. URL: https://segment-anything.com/ (cit. on p. 34).

[17] Anurag Ghosh. *Segment Anything Labelling Tool (SALT)*. 2023. URL: https://github.com/anuragxel/salt (cit. on p. 34).

[18] Diederik P. Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *CoRR* abs/1412.6980 (2014). URL: https://api.semanticscholar.org/CorpusID:6628106 (cit. on p. 35).

[19] Ilya Loshchilov and Frank Hutter. «Fixing Weight Decay Regularization in Adam». In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101 (cit. on p. 36).

[20] Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, and Peter Richtárik. «SGD: General Analysis and Improved Rates». In: *CoRR* abs/1901.09401 (2019). arXiv: 1901.09401. URL: http://arxiv.org/abs/1901.09401 (cit. on p. 37).

[21] G. Bradski. «The OpenCV Library». In: *Dr. Dobb's Journal of Software Tools* (2000) (cit. on p. 47).

[22] Steven M LaValle. *Rapidly-exploring random trees: A new tool for path planning*. Tech. rep. Computer Science Department, Iowa State University, 1998 (cit. on p. 56).

[23] Steven M LaValle. *Planning algorithms*. Cambridge University Press, 2006 (cit. on p. 56).

[24] James J Kuffner and Steven M LaValle. «RRT-Connect: An efficient approach to single-query path planning». In: *Proceedings of the IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE. 2000, pp. 995–1001 (cit. on p. 57).

[25] Lydia E Kavraki, Petr Švestka, Jean-Claude Latombe, and Mark H Overmars. «Probabilistic roadmaps for path planning in high-dimensional configuration spaces». In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580 (cit. on p. 57).

[26] Matt Zucker, Nathan Ratliff, Anca D Dragan, Michael Pighetti, Matthew Klingensmith, Chris M Dellin, J Andrew Bagnell, and Siddhartha S Srinivasa. «CHOMP: Covariant Hamiltonian optimization for motion planning». In: *The International Journal of Robotics Research* 32.9-10 (2013), pp. 1164–1193 (cit. on p. 58).

[27] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. «STOMP: Stochastic trajectory optimization for motion planning». In: *Proceedings of the IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 4569–4574 (cit. on p. 59).

[28] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. «Optuna: A Next-generation Hyperparameter Optimization Framework». In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019 (cit. on pp. 61, 80).