DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Safety-Oriented Task Offloading for Human-Robot Collaboration

A Learning-Based Approach

FRANCO RUGGERI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Safety-Oriented Task Offloading for Human-Robot Collaboration

A Learning-Based Approach

FRANCO RUGGERI

Degree Programme in Computer Science and Engineering
Date: October 25, 2021

Supervisors: Iolanda Leite, Tatiana Tommasi
Industrial Supervisors: Rafia Inam, Alberto Hata, Ahmad Terra
Examiner: Danica Kragic Jensfelt
School: School of Electrical Engineering and Computer Science
Host company: Ericsson AB
Swedish title: Säkerhetsorienterad Uppgiftsavlastning för Människa-robotkollaboration
Swedish subtitle: Ett Inlärningsbaserat Tillvägagångssätt

Abstract

In Human-Robot Collaboration scenarios, safety must be ensured by a risk management process that requires the execution of computationally expensive perception models (e.g., based on computer vision) in real-time. However, robots usually have constrained hardware resources that hinder timely responses, resulting in unsafe operations. Although Multi-access Edge Computing allows robots to offload complex tasks to servers on the network edge to meet real-time requirements, this might not always be possible due to dynamic changes in the network that can cause congestion or failures. This work proposes a safety-based task offloading strategy to address this problem. The goal is to intelligently use edge resources to reduce delays in the risk management process and consequently enhance safety. More specifically, depending on safety and network metrics, a Reinforcement Learning (RL) solution is implemented to decide whether a less accurate model should run locally on the robot or a more complex one should run remotely on the network edge. A third possibility is to reuse the previous output through verification of temporal coherence. Experiments are performed in a simulated warehouse scenario where humans and robots have close interactions. Results show that the proposed RL solution outperforms the baselines in several aspects. First, the edge is used only when the network performance is good, reducing the number of failures (up to $47\,\%$). Second, the latency is also adapted to the safety requirements (*risk* $\times$ *latency* reduced up to $48\,\%$), avoiding unnecessary network congestion in safe situations and letting other robots in hazardous situations use the edge. Overall, the latency of the risk management process is largely reduced (up to $68\,\%$), and this positively affects safety (time in safe zone increased up to $3.1\,\%$).

Keywords

Human-Robot Collaboration, Multi-access Edge Computing, Task Offloading, Artificial Intelligence, Reinforcement Learning

Sammanfattning

I scenarier med människa-robotkollaboration måste säkerheten säkerställas via en riskhanteringsprocess. Denna process kräver exekvering av beräkningstunga uppfattningsmodeller (t.ex. datorseende) i realtid. Robotar har vanligtvis begränsade hårdvaruresurser vilket förhindrar att respons uppnås i tid, vilket resulterar i osäkra operationer. Även om Multi-access Edge Computing tillåter robotar att avlasta komplexa uppgifter till servrar på edge, för att möta realtidskraven, så är detta inte alltid möjligt på grund av dynamiska förändringar i nätverket som kan skapa överbelastning eller fel. Detta arbete föreslår en säkerhetsbaserad uppgiftsavlastningsstrategi för att hantera detta problem. Målet är att intelligent använda edge-resurser för att minska förseningar i riskhanteringsprocessen och följaktligen öka säkerheten. Mer specifikt, beroende på säkerhet och nätverksmätvärden, implementeras en Reinforcement Learning (RL) lösning för att avgöra om en modell med mindre noggrannhet ska köras lokalt eller om en mer komplex ska köras avlägset på edge. En tredje möjlighet är att återanvända sista utmatningen genom verifiering av tidsmässig koherens. Experimenten utförs i ett simulerat varuhusscenario där människor och robotar har nära interaktioner. Resultaten visar att den föreslagna RL-lösningen överträffar baslinjerna i flera aspekter. För det första används edge bara när nätverkets prestanda är bra, vilket reducerar antal fel (upp till $47\,\%$). För det andra anpassas latensen också till säkerhetskraven (*risk* $\times$ *latens* reducering upp till $48\,\%$), undviker onödig överbelastning i nätverket i säkra situationer och låter andra robotar i farliga situationer använda edge. I det stora hela reduceras latensen av riskhanterings processen kraftigt (upp till $68\,\%$) och påverkar på ett positivt sätt säkerheten (tiden i säkerhetszonen ökas upp till $4\,\%$).

Nyckelord

Människa-robotkollaboration, Multi-access Edge Computing, Uppgiftsavlastning, Artificiell intelligens, Reinforcement Learning

Acknowledgments

The completion of this thesis project represents the end of a path from which I come out very grown, not only from the knowledge perspective but especially as a person. Without a doubt, one of the most significant contributions to this growth is my experience in Stockholm and at Ericsson. For this reason, I would like to express my sincere gratitude to my supervisors Rafia Inam, Alberto Hata, and Ahmad Terra from Ericsson. They allowed me to work on this exciting project in a truly global company where everyone has a positive mindset directed towards innovation. Beyond this, they provided me constant support and valuable suggestions during our weekly meetings. Likewise, I would like to thank my academic supervisors Iolanda Leite and Tatiana Tommasi, for always being available to clear my doubts.

Stockholm, October 2021
Franco Ruggeri

Contents

List of Figures

List of Tables

Algorithms

List of acronyms and abbreviations

A2C Advantage Actor-Critic

AI Artificial Intelligence

AP access point

API application programming interface

BS base station

CBR constant bit-rate

CC Cloud Computing

CEM Cross-Entropy Method

CNN Convolutional Neural Network

CPU Central Processing Unit

CR Cloud Robotics

CSMA Carrier-Sense Multiple Access

CV computer vision

DDR Double Data Rate

DL Deep Learning

DNN Deep Neural Network

DQN Deep Q-Network

DRL Deep Reinforcement Learning

EC Edge Computing

FL Fuzzy Logic

FOV field of view

FPS frames per second

GD gradient descent

GPU Graphics Processing Unit

H-D Hypothetico-Deductive

HAZOP HAZard OPerability analysis

HRC Human-Robot Collaboration

ICMP Internet Control Message Protocol

KPI Key Performance Indicator

LAN Local Area Network

LiDAR Light Detection and Ranging

MD mobile device

MDP Markov Decision Process

MEC Multi-access Edge Computing

ML Machine Learning

MSS Maximum Segment Size

MTU Maximum Transmission Unit

NE Nash equilibrium

NIC Network Interface Card

ns-3 Network Simulator 3

OS operating system

PC personal computer

RAM Random Access Memory

RAN Radio Access Network

ReLU Rectified Linear Unit

RGB Red Green Blue

RL Reinforcement Learning

RNG Random Number Generator

ROS Robotic Operating System

RPC remote procedure call

RTT Round-Trip Time

SARSA State–Action–Reward–State–Action

SINR Signal-to-Interference-Noise Ratio

SSD Solid-State Drive

SSIM Structural Similarity Index Measure

TCP Transmission Control Protocol

UDP User Datagram Protocol

V-REP Virtual Robot Experimentation Platform

Wi-Fi Wireless Fidelity

YANS Yet Another Network Simulator

Chapter 1

Introduction

This chapter describes the specific problem that this thesis addresses and introduces the context, the scope, and the goals of this degree project. In addition, it outlines the structure of the thesis.

1.1 Background

The recent improvements in robotics and machine learning have enabled a new paradigm for working environments, where humans and robots share the same workspace and collaborate to perform tasks. This approach, known as Human-Robot Collaboration (HRC), represents a fundamental aspect in industry 4.0 and allows to improve performance and quality of the industrial production [4]. The reason for this enhancement is related to the unique features of robots and humans: robots outperform humans when it comes to repetitive tasks requiring power, precision, and speed, but humans are creative, more intelligent, and more flexible [5]. However, while HRC is beneficial, sharing the same workspace with possible close interactions between humans and robots raises new challenges, among which the most relevant is related to safety: robots must minimize the risk of harming humans.

Recent research [5, 6, 7, 8, 9] proposed and implemented a safety framework, shown in figure 1.1, for mobile robots in automated warehouses with HRC. While this solution aims to perform on each robot a risk management process in real-time, it also involves the execution of a computationally expensive computer vision (CV) task as the first step, namely the scene understanding. Unfortunately, mobile robots often have cheap onboard hardware limiting energy consumption and size to operate longer and maneuver easily, which is insufficient for this complex task. Terra et

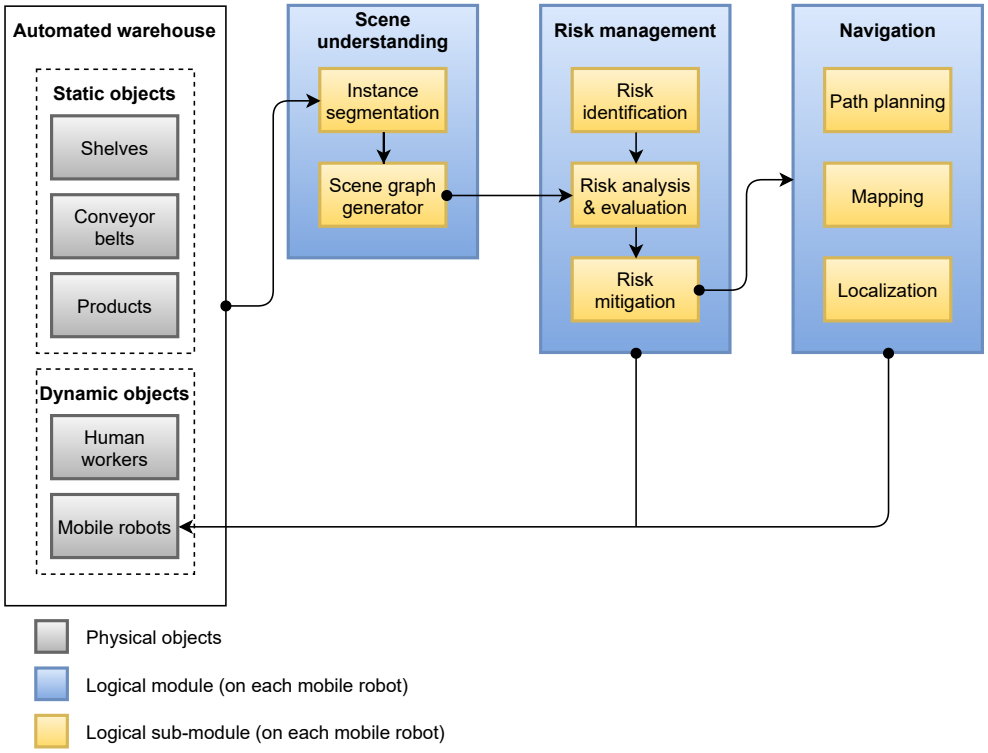


Figure 1.1: Safety framework for HRC in automated warehouses.

al. [8] showed that the scene understanding has unacceptable delays when executed on a mobile robot, resulting in unsafe operations. To overcome this issue, the authors proposed a distributed architecture with the scene understanding performed on an edge device, shown in figure 1.2. Indeed, Multi-access Edge Computing (MEC) is a promising solution that enables resource-constrained mobile devices (MDs) to execute computation-intensive, time-sensitive applications by offloading them to servers located at the edge of the network, close to the end-users [10, 11]. However, due to dynamic changes in the network that can cause congestion or failures, using the edge is not always possible or suitable. In particular, in case of network congestion or failures, the risk management process keeps waiting for the output from the scene understanding, leading to unsafe operations. It is worth noticing that congestion is unavoidable when many mobile robots continually offload the scene understanding because the inputs sent on the network are large. For this reason, the static solution of figure 1.2, which always offloads the scene understanding, is not adequate.

This project proposes a safety-oriented solution that, depending on safety and network Key Performance Indicators (KPIs), dynamically decides whether

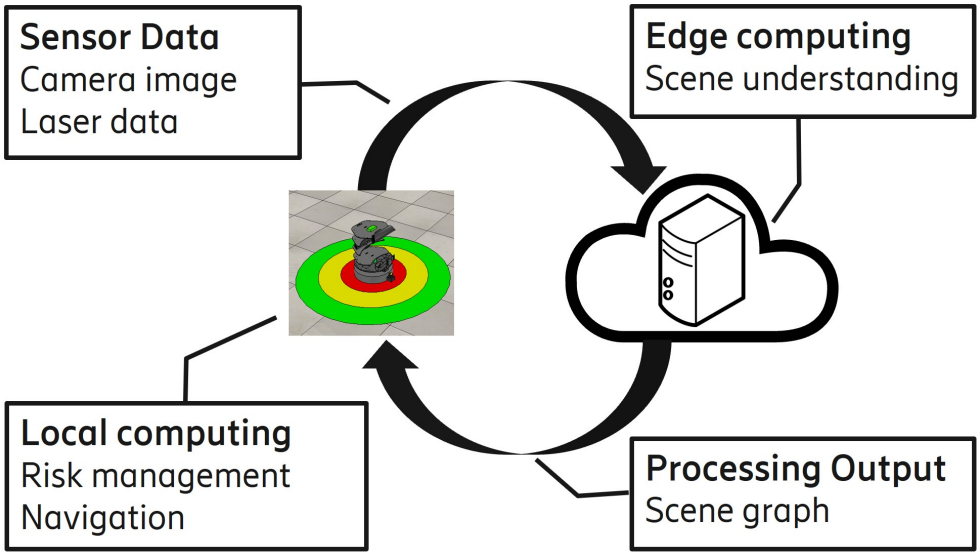


Figure 1.2: Distributed architecture with the scene understanding always offloaded to an edge server.

the complex scene understanding should be offloaded to an edge server or a less accurate model should run locally. A third possibility is to skip the computation and reuse the last output, exploiting the temporal coherence in sequential images. The main goal is to use the edge only when suitable, enhancing safety in the warehouse while optimizing the usage of the network resources. Reinforcement Learning (RL) is used to learn policies from the environment, instead of manually writing rules.

The work was carried out at the Ericsson AB department of Research, AI, and Cognitive Automation Lab. The proposed method was filed on 26/02/2021 as a patent idea: *Intelligent Task Offloading*.

1.2 Problem

In an automated warehouse with HRC, mobile robots can guarantee safety by using the safety framework implemented in [5, 6, 7, 8, 9]. The first step of the process, i.e., the scene understanding, is a computationally expensive, latency-critical, safety-critical task and needs to exploit MEC to fulfill the latency requirements. However, networks might not always be able to satisfy the real-time requirements due to dynamic changes that can cause congestion or failures. It is necessary to find policies to decide whether to offload or not dynamically, based on network and safety KPIs.

The research question that will be addressed in this thesis is: *how can mobile robots intelligently offload to the edge their computationally expensive, latency-critical, safety-critical tasks to maximize safety in HRC scenarios?*

This question can be broken down into the following sub-questions, which can be more easily investigated:

- *Which actions are possible when offloading is not suitable or possible for computationally expensive, latency-critical, safety-critical task?*
- *Which KPIs are useful for the safety-oriented task-offloading decision-making?*
- *Which RL algorithm can be used to learn safety-oriented task-offloading policies?*
- *What is the impact of safety-oriented task-offloading policies on safety?*
- *What is the impact of safety-oriented task-offloading policies on the usage of network resources?*

1.3 Purpose

This degree project is built on top of [5, 6, 7, 8, 9], whose purpose is to create a risk management strategy for HRC in automated warehouses. The success of this work represents a further step towards a safe HRC in industry 4.0, providing benefits to both industrial employers and employees. Indeed, HRC has a great impact both on the industrial work chain, affecting the financial aspects positively, and on the employees' work experience because robots can perform repetitive tasks. Furthermore, safety helps humans to trust robots, as they do not feel afraid of getting injured.

In order to guarantee ethical research, all the methods and results were backed up with scientific evidence. In particular, the results can be reproduced using the code stored on the Ericsson AI GitLab server.

1.4 Goals

The main goal of this degree project is to *propose a safety-oriented, learning-based solution to the task-offloading problem for HRC scenarios*. This goal can be divided into the following sub-goals:

- *Design and implement a modular system to integrate the new task-offloading module with the existing modules.*
- *Design and implement an RL solution to learn safety-oriented task-offloading policies.*
- *Design and implement a simulation of a warehouse with a network.*
- *Evaluate the solution using meaningful evaluation metrics and baselines.*

1.5 Research methodology

This thesis uses the *Hypothetico-Deductive (H-D) method* and the *experimental methodology*. The handled data is mainly primary and is collected from simulations, but some results from previous work are used to make assumptions and set parameters. The results are presented using both quantitative and qualitative data.

The experiments were completely performed in simulated environments with the help of Virtual Robot Experimentation Platform (V-REP) [12, 13] for the warehouse and Network Simulator 3 (ns-3) [14, 15] for the network, which model the real world very accurately. Such an approach leads to a more reliable evaluation compared to experiments with synthesized data that consider ideal behavior and few phenomena, such as those carried out by many previous works in MEC. In general, simulations have been recognized as a very important research tool [16]: in HRC scenarios, they allow to study and design new functions without the risk of harming real humans; in networks, they can reproduce complex systems with many physical phenomena without the difficulty of setting up costly testbeds [17]; in RL, agents are often entirely trained in a simulated environment before being deployed [18].

There is another non-technical reason why simulations were preferred to real-world experiments, i.e., with real robots and a real network: unfortunately, this project was carried out during the COVID-19 pandemic and, for safety reasons, from home.

1.6 Delimitations

This project uses the safety framework implemented in [5, 6, 7, 8, 9]. It is worth remarking that these modules were already available, and a modification

of them is out of the scope of this work. On the other hand, it was necessary to tune parameters and make some changes to get everything working in the new warehouse scenario (e.g., navigation). In addition to this, the designed solution considers a fast instance segmentation model installed on the robot as an alternative to the complex and accurate model installed on the MEC server. While research of efficient models for MDs was necessary for this work to perform meaningful experiments, training and using those models for the automated warehouse is out of the scope of this thesis and is left as future work.

There are also some limitations due to hardware resources. Since the entire simulation, including V-REP and ns-3, has to be run in real-time on one computer, dealing with performance is a big issue. Specifically, if the simulation is too heavy, ns-3 cannot keep real-time, and V-REP runs at a meager frame rate. Such concerns affected several choices for the experiments:

- The number of robots, simulated both physically in V-REP and as network nodes in ns-3, was limited to two. The congestion in the network was increased utilizing congesting nodes, i.e., virtual devices simulated only as network nodes in ns-3.
- The network technology was set to Wi-Fi 802.11g [19] and not to more capable ones that would have been heavier to congest (e.g., Wi-Fi 802.11n [20]).
- An alternative implementation of the scene understanding extracting the output by querying V-REP was used, and the execution latency was simulated by sleeping for a certain time. This workaround represents a good solution in terms of latency but does not simulate the different accuracy of local and edge computing.

1.7 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 introduces the technical background necessary to understand the rest of the thesis, the previous work this project is built on, and the related work in the literature. Chapter 3 presents the methods used to solve the problem and the implementation. Chapter 4 discusses the experiments and analyzes the results. Lastly, chapter 5 gives the conclusions and suggests future work.

Chapter 2

Background

This chapter presents the related work and the technical background needed to understand the rest of the thesis. Section 2.1 provides details about the previous work in HRC representing the basis of this project, which has been briefly introduced in chapter 1. Section 2.2 discusses the high-level ideas of MEC and how it was used in HRC, focusing on the problems. Section 2.3 briefly presents the background on RL, with particular attention on Deep Q-Network (DQN) which is the algorithm selected for the solution. Section 2.4 provides the main concepts of the frameworks and tools used for the implementation without claiming to be exhaustive. Lastly, section 2.5 reviews the literature in MEC, highlighting limitations and useful ideas reused in this degree project.

2.1 Human-Robot Collaboration

Human-Robot Collaboration (HRC) is an emerging research area that studies collaborative processes in which humans and robots cooperate to achieve shared goals. Industrial applications can benefit from it by combining robots' power, speed, and precision with the creativity and flexibility of humans to allow more versatile automation steps that improve productivity and quality [4]. However, HRC causes new hazardous situations, and therefore collaborative robots require proper safety strategies to avoid harming humans and their environments [5].

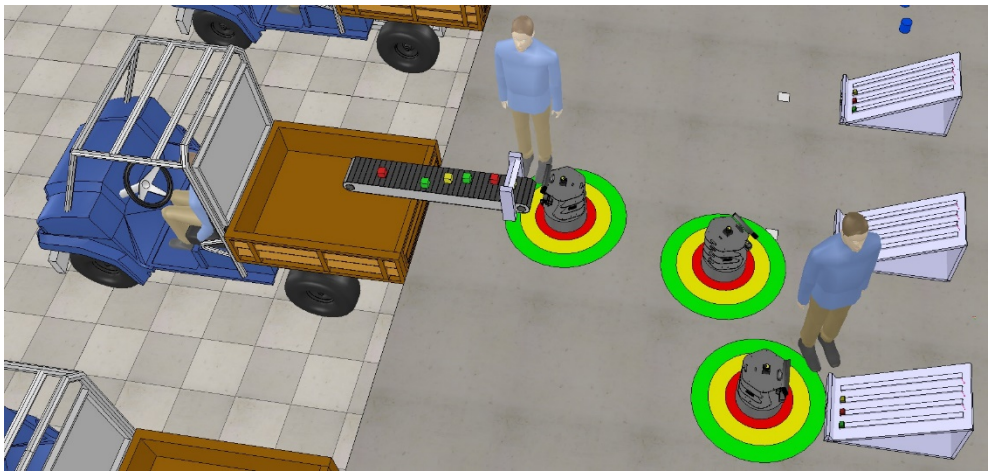


Figure 2.1: Warehouse simulated with V-REP in which human workers and Turtlebot2i robots collaborate to load trucks.

2.1.1 Safety in automated warehouse

Recent research work [5, 6, 7, 8, 9] was carried out for automated warehouses where mobile robots and humans collaborate to load trucks. In this scenario, the robots pick products from shelves and place them on conveyor belts, while the human workers fill the shelves with the ordered products. Robots and humans are in the same workspace during such operations and can have close interactions, especially around the shelves.

Inam et al. [5] proposed a two-fold safety strategy for this scenario, including:

- *Offline safety analysis*: A warehouse controller evaluates the high-level plans for the robots before sending the tasks to each robot.
- *Online safety analysis*: Each robot generates three safety fields around itself, whose size is taken from standards (e.g., ISO/TS 15066:2016). The safety fields are critical (red), warning (yellow), and safe (green). Then, it analyzes its sensor data to assess and mitigate the risk. This process is repeated continually at run-time.

In subsequent works [6, 7, 8, 9], the online safety analysis was further detailed as the safety framework shown in figure 1.1, focusing on the navigation phase[1], whereas not considering the manipulation one[2]. The

[1] The robot's base moves towards waypoints while the arm is tucked in.
[2] The robot's arm picks or places products while the base is still.

implementation was done for a Turtlebot 2i [21, 22] using Robotic Operating System (ROS) [23] and the experiments were conducted in simulated warehouses using V-REP [12]. Figure 2.1 shows an example of a scenario with human workers and Turtlebot 2i robots with their safety fields.

2.1.2 Risk management process

The core of the safety framework in figure 1.1 is the risk management process, which consists of the following phases:

1. *Risk identification*: The first phase is to identify the possible hazards and risks that can happen in the scenario. It was done manually using HAZard OPerability analysis (HAZOP) and is the only phase not performed at run-time [6].

2. *Risk analysis and evaluation*: The second phase analyzes the surroundings and evaluates how risky the situation is. The input is a scene graph, which models the robot's knowledge of the surroundings and is computed by the scene understanding module, as discussed in section 2.1.3, while the output is a risk value for each node in the scene graph. Furthermore, the three-layered safety fields are generated based on the risk values and other information (e.g., robot's speed). This module was implemented using a Fuzzy Logic (FL) system [7].

3. *Risk mitigation*: The last phase uses the scene graph and the risk values to reduce the risk and allow safe navigation. In particular, this is done by scaling the speed of the robot's left and right wheels to adjust linear and angular velocities given by the navigation module[3]. Two methods were used for the implementation of this module: FL and DQN [8].

It is worth mentioning that the risk management process needs to be executed in real-time. In other words, it is a safety-critical, latency-critical task. Indeed, the risk mitigation has to be executed frequently and with updated information to adjust the navigation trajectory effectively.

2.1.3 Scene understanding

The scene understanding provides the necessary input to the risk management process. Specifically, it analyzes sensor data to generate a scene graph

[3] Turtlebot 2i is a differential drive robot, as described in section 2.4.6. Thus, both linear and angular velocities can be modified by controlling the speed of the left and right wheels.

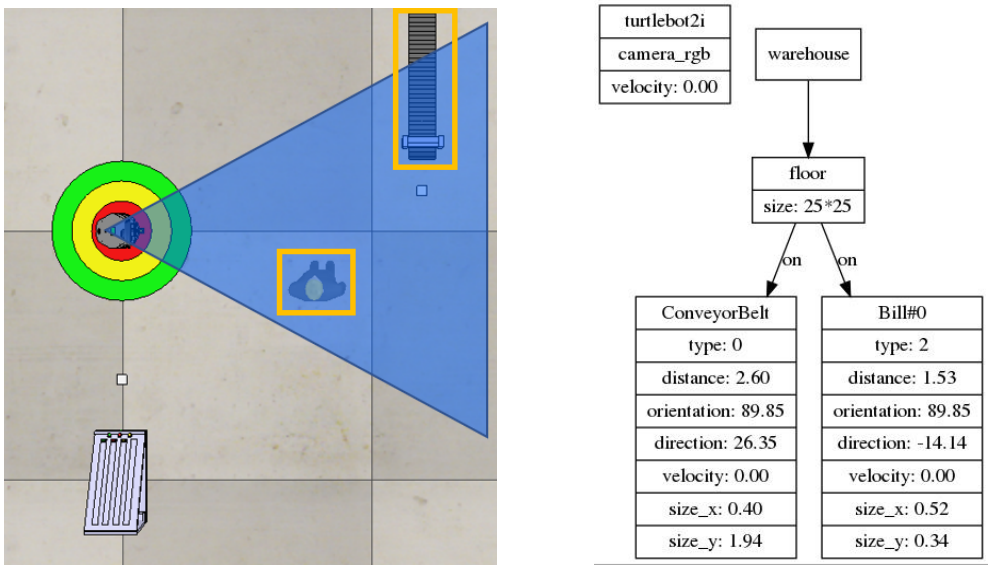


Figure 2.2: A Turtlebot2i detects the objects in its FOV and builds the scene graph.

modeling the objects in the robot's field of view (FOV) and their relationships. As shown in figure 2.2, each graph node denotes an object and contains semantic information such as the type (0 for static objects, 1 for dynamic objects, 2 for humans), while a graph edge gives the position relationship between two objects.

The computation of the scene graph is done in two steps:

1. Instance segmentation: the RGB camera image is processed to detect and identify the objects. This module was implemented as a Mask R-CNN with ResNet101 backbone, a state-of-the-art Convolutional Neural Network (CNN) for accurate instance segmentation [9, 24].

2. Scene graph generation: the detected objects are organized in a scene graph. Moreover, other sensor data is used to compute semantic information of the objects. For example, the depth camera image is used to find the distance of the objects from the robot.

It is essential to highlight that the scene understanding is a safety-critical, latency-critical task because it has to provide timely output to the risk management process. Moreover, it is also computationally expensive because of the enormous number of parameters of the Mask R-CNN.

An alternative implementation of the scene understanding was developed by extracting the scene graph directly from V-REP. Although this solution

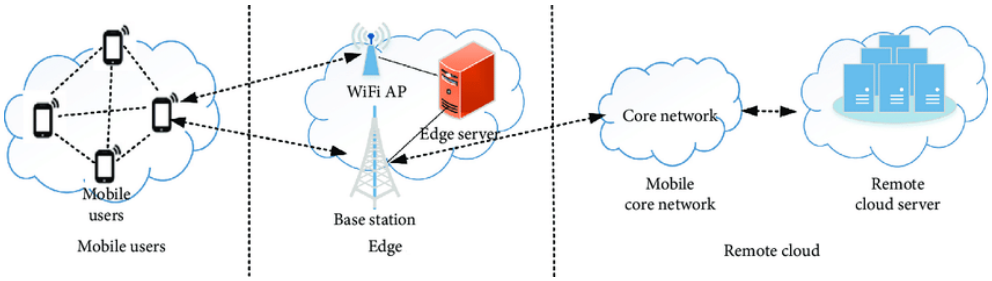


Figure 2.3: MEC architecture [1].

is not applicable in real environments, it turns out to be very useful for experiments in simulations because it allows isolating the module to test without depending on the accuracy and delay of the instance segmentation. For example, it was used for testing the risk mitigation algorithms [8], and it was used for the experiments of this project, as described in chapter 3.

2.2 Multi-access Edge Computing

In the last decade, Cloud Computing (CC) has emerged as a new computing paradigm providing centralized computational resources and data storage to the end-users employing large-scale data centers [25]. Although being used by all kinds of machines, it comes especially in handy for resource-constrained devices such as MDs (e.g., smartphones) and robots, which can offload computation-intensive tasks to servers on the cloud. Indeed, a popular application of CC is in robotics and is known as Cloud Robotics (CR) [26, 27]. However, due to the large distance of the data centers from the end-users, CC cannot guarantee very low latency and thus is not suitable for real-time tasks.

In recent years, Edge Computing (EC) has been proposed as solution to considerably reduce latency compared to CC, allowing end users to offload also computation-intensive tasks with low-latency requirements [10, 11]. The most common application is with mobile terminals in Radio Access Networks (RANs) (4G and 5G) or wireless LANs (Wi-Fi), as shown in figure 2.3, in which case EC is better known as Multi-access Edge Computing (MEC)[4]. MEC servers are deployed very close to the end users, at the edge of the network, either directly on base stations (BSs) and access points (APs), or in small-scale data centers very close to these.

It is worth remarking that CC and MEC are not alternatives; instead, they are complementary because they provide a different trade-off between latency

[4] Formerly Mobile Edge Computing.

and computation power. The choice depends on the requirements of the task.

2.2.1 MEC for safety framework in HRC

Experiments by Terra et al. [8] showed that the only computationally expensive task in the safety framework described in section 2.1 is the scene understanding with Mask R-CNN. In particular, with local computation on a Turtlebot 2i, all the phases of the risk management process run with ultra-low execution times, in the order of milliseconds, whereas the scene understanding takes almost 15 seconds. Such considerable delays in the scene understanding represent a bottleneck for the whole safety framework and lead to unsafe operations.

Terra et al. proposed to offload the scene understanding to an edge server, as shown in figure 1.2, to achieve real-time performance. They tested this approach in a network with only one robot. However, this static solution is not robust to dynamic changes in the network, which can cause congestion or failures, and this is the motivation for this degree project. In these cases, the robot may not get the output from the scene understanding, resulting in the risk management process waiting and not providing safety. In addition to this, when many mobile robots use this solution, network congestion is unavoidable because the scene understanding offloading involves sending large inputs on the network. Therefore, dynamic decision-making to intelligently use MEC is necessary. Many strategies to address this problem are available in the literature and are discussed in section 2.5.

2.3 Reinforcement Learning

Reinforcement Learning (RL) [28] is the subfield of Machine Learning (ML) that studies how to learn from interaction with uncertain environments. Such interaction is illustrated in figure 2.4: the agent takes an action based on the current environment state and receives a feedback in the form of a numerical reward; also, the agent senses the environment again to get its new state in order to repeat the loop. In this setting, the agent must learn to act in order to maximize the expected rewards. It is worth highlighting that RL deals with sequential decision-making and trial-and-error learning. The former means that the action usually affects not only the immediate reward but also the next environment state and, through that, all the subsequent rewards. The latter indicates that the agent must discover which actions yield the most reward by

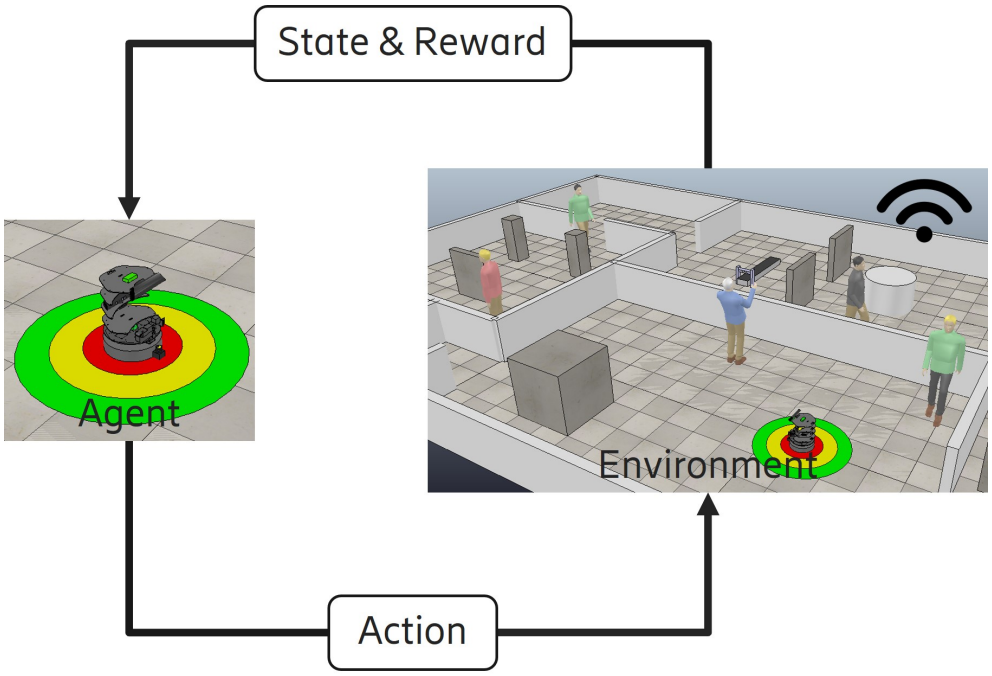


Figure 2.4: RL agent-environment interaction in automated warehouse.

trying them out, unlike supervised learning in which it is instructed with the correct action (the label).

The RL framework is very general and can be applied for all the problems involving making a sequence of decisions. Indeed, RL has been widely used with good results in many areas, such as robotics [29], healthcare [30], and game playing [31].

2.3.1 Markov Decision Process

A general[5] RL problem is fully defined by the environment, which can be formalized as a Markov Decision Process (MDP). A Markov Decision Process (MDP) is a discrete-time stochastic control process modeled as a 5-tuple (S, A, T, R, γ) where:

- S is the state space, i.e., the set of possible environment states.
- A is the action space, i.e., the set of possible actions that the agent can perform on the environment.

[5] RL problems satisfying specific constraints can be formalized more simply as multi-armed or contextual bandits, but they are omitted as not relevant for this work.

- $T : S \times A \times S \rightarrow [0, 1]$ is the state transition function giving the probability that action $a \in A$ in state $s \in S$ leads to the new state $s' \in S$, i.e., $T(s, a, s') = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a)$.
- $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function giving the numerical reward for the agent when action $a \in A$ in state $s \in S$ leads to the new state $s' \in S$, i.e. $R(s, a, s') \in \mathbb{R}$. Often the reward function R is independent of the action $a \in A$ or the new state $s' \in S$, or both of them.
- $\gamma \in [0, 1]$ is the discount factor used to compute the return with an exponentially decay of future rewards, as can be seen in equation (2.2). This is necessary to compare policies in infinite-horizon problems[6].

MDPs have the Markov property, which generally means that future and past are independent given the present. In mathematical terms:

$$\mathbb{P}\left(s_{t+1} | s_t, a_t\right) = \mathbb{P}\left(s_{t+1} | s_t, a_t, ..., s_0, a_0\right) \tag{2.1}$$

A solution to a MDP is a policy[7] $\pi : S \rightarrow A$ expressing which action $a \in A$ to choose for each state $s \in S$. RL aims to find an optimal policy π^*, i.e., a policy that maximizes the expected return for all the states $s \in S$. The expected return, also known as V-value function or state-value function, estimates how good a state $s \in S$ is under a policy π:

$$V_\pi\left(s\right) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, \pi\right] \tag{2.2}$$

In RL T and R are unknown, so the agent must try actions to learn. This key feature also leads to a key challenge known as the exploration-exploitation dilemma: the agent has to make a trade-off between exploiting the optimal policy so far and exploring different actions, which might turn out to be better than the current optimal ones. A widespread solution, also used in this project, is the ϵ-greedy policy, which acts randomly with probability ϵ and according to the optimal policy with probability $1 - \epsilon$.

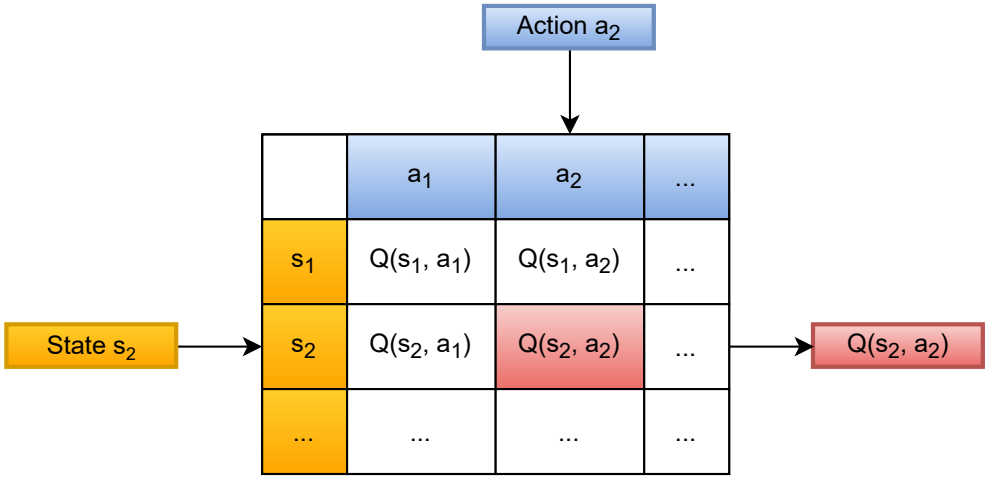


Figure 2.5: Q-learning stores the Q-values for all the state-action pairs in the Q-table.

2.3.2 Q-learning

A very popular RL algorithm is Q-learning [32]. Instead of the V-value function, it uses the Q-value function, also known as action-value function, that estimates the value of taking action $a \in A$ in state $s \in S$ and thereafter following the policy π:

$$Q_\pi(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \middle| S_t = s, A_t = a, \pi\right] \tag{2.3}$$

The optimal Q-value function can be written recursively using the Bellman equation and is independent of the policy:

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q_\pi(s,a) \\ &= \mathbb{E}\left[R_t + \gamma \max_{a' \in A} Q^*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \end{aligned} \tag{2.4}$$

Q-learning aims to approximate Q^* incrementally with the following update:

$$Q(S_t, A_t) \leftarrow (1-\alpha) Q(S_t, A_t) + \alpha \left[R_t + \gamma \max_{a \in A} Q(S_{t+1}, a)\right] \tag{2.5}$$

[6] Problems that go on continually without limit and cannot naturally break into episodes.

[7] A policy can potentially also be stochastic, but here only deterministic policies are considered for simplicity.

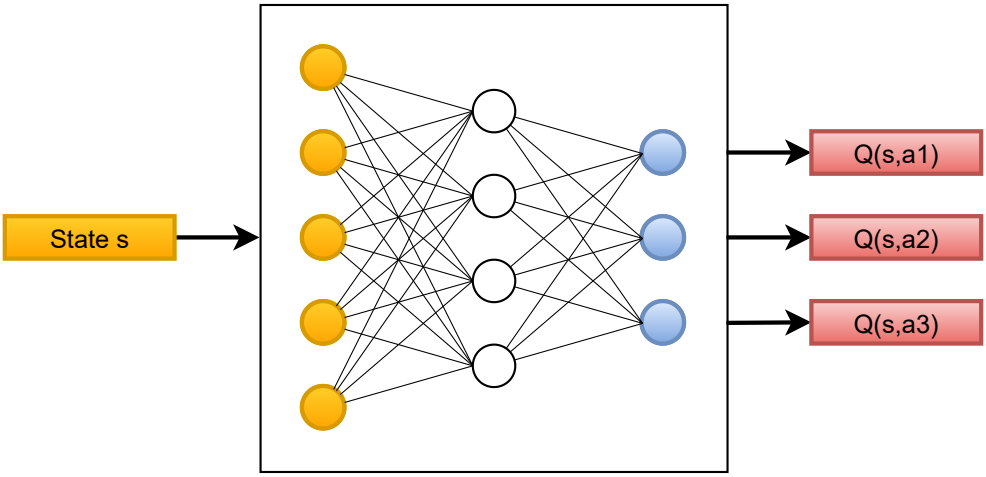


Figure 2.6: DQN approximates the Q-value function with a DNN. The input layer corresponds to the features of the state, while the output layer contains one neuron for each possible action.

where α is the learning rate. Given Q^*, the optimal policy is:

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a) \tag{2.6}$$

Q-learning is proved to converge to the optimal Q-value function provided that all the actions are repeatedly sampled in all the states (i.e., there is sufficient exploration), independently of the policy used[8]. This condition can be guaranteed by using the ϵ-greedy policy, described in the previous subsection, which also addresses the exploration-exploitation dilemma.

The main limitation of Q-learning is that it needs to store the Q-values for all the state-action pairs in a lookup table called Q-table, as illustrated in figure 2.5. Consequently, state and action spaces must be discrete and low-dimensional, which is not the case for many real-world problems. DQN overcomes this constraint.

2.3.3 Deep Q-Network

Deep Reinforcement Learning (DRL) [33] combines RL and Deep Learning (DL) to overcome the limitations due to high-dimensional state and action spaces. The Deep Reinforcement Learning (DRL) extension of Q-learning is called Deep Q-Network (DQN) and became popular for outperforming human

[8] This is known as off-policy learning.

experts in several Atari games by learning directly from pixels [34].

The main idea is to parameterize the Q-value function with a Deep Neural Network (DNN), as shown in figure 2.6. Such a DNN receives the state as input and outputs the Q-values for all the possible actions. In this way, DQN can relax the limitation of Q-learning and accommodate high-dimensional, even continuous, state spaces. The problem is still to find the optimal Q-value function, but it is achieved by training the DNN as follows:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(Y_t^Q, \theta)\Big|_{\theta_t} \tag{2.7}$$

where θ represents the weights of the DNN, L is a loss function (e.g., squared loss), and Y_t^Q is the target Q-value at the t-th step. According to equation (2.4) and equation (2.5), Y_t^Q is:

$$Y_t^Q = R_t + \gamma \max_{a \in A} Q\left(S_{t+1}, a; \theta_t\right) \tag{2.8}$$

However, DNNs work well when there is a fixed dataset providing stable targets, while Y_t^Q in equation (2.8) depends on θ which is continually updated. This might cause instabilities and divergence. To solve the problem, DQN uses a second DNN, known as target network, to generate the target:

$$Y_t^Q = R_t + \gamma \max_{a \in A} Q\left(S_{t+1}, a; \theta_t^-\right) \tag{2.9}$$

where θ^- represents the weights of the target network and is updated periodically every $C \in \mathbb{N}$ steps with the weights of the original network, i.e., $\theta^- = \theta$. The period C has to be large enough to guarantee convergence.

Another trick used by DQN to improve stability and convergence is the replay memory. Basically, the agent stores the experience of the last $N \in \mathbb{N}$ steps as tuples (S_t, A_t, S_{t+1}, R_t). In this way, the DNN can be trained with mini-batch gradient descent (GD) by randomly selecting mini-batches from the replay memory. Mini-batch GD has many advantages over online GD, such as more accurate estimates of the gradients and efficient parallelization of the algorithm.

The solution developed in this project uses DQN because, as will be described in chapter 3, the designed RL environment has a continuous state space.

Figure 2.7: ROS topic with one publisher and two subscribers [2].

2.4 Frameworks and tools

The solution proposed in this degree project was implemented using standard frameworks and popular tools. ROS was used to distribute the code on robot and MEC server, as well as to integrate it with other available modules and packages. The simulation was realized using V-REP for the physical environment (e.g., robots and objects in the warehouse) and ns-3 for the network, with ROS integrating the two simulators. The RL environment was implemented following the OpenAI Gym API so as to use Keras-RL for the RL agent. Lastly, following the previous work described in section 2.1, the implementation was done for a Turtlebot 2i. The rest of this section provides an overview of these frameworks and tools along with the terminology necessary to understand chapter 3, which contains more information on how they are used in this project.

2.4.1 ROS

Robotic Operating System (ROS) [23, 35] is the most popular open-source framework for writing robot software. Despite the name, it is not an operating system (OS) in the traditional sense but runs above a host OS. The main idea is to distribute the tasks of a robot system in several processes and provide structured communication to exchange data among them. For instance, one process can perform path planning, one can perform localization, and another can control the wheel motors. Such processes can also run on different machines, and this is a handy functionality for CR and MEC.

ROS represents processes and their communication as nodes and edges, respectively, in a computation graph. In this regard, the main concepts are [36]:

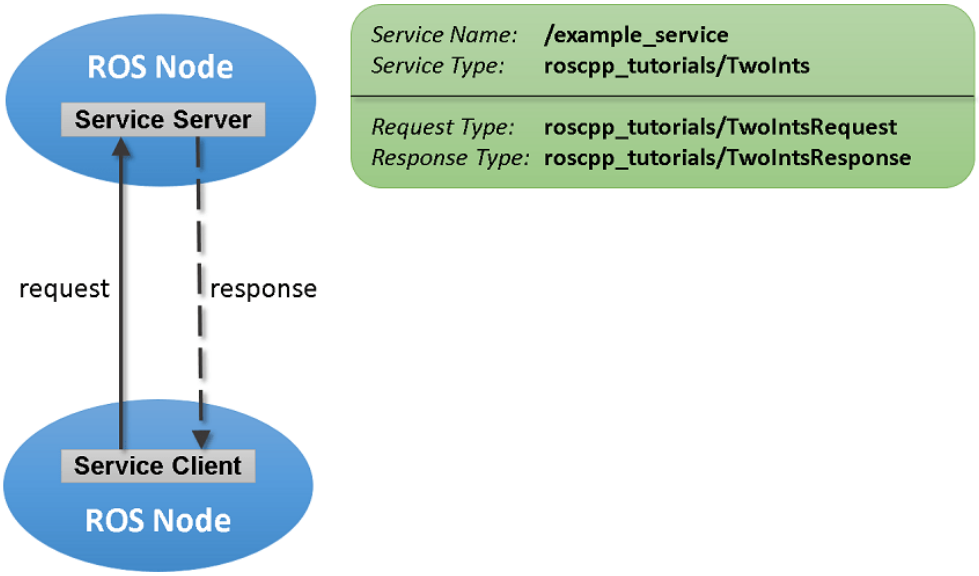


Figure 2.8: ROS service [3].

- *Node*: A ROS node is a process performing computation which is coded with a ROS client library, i.e., either roscpp in C++ [37] or rospy in Python [38].
- *Message*: A ROS message is basically a data structure and is defined in simple text files. ROS nodes can communicate with each other by passing ROS messages.
- *Topic*: A ROS topic transports ROS messages of a certain type between ROS nodes with a many-to-many paradigm. Thus, there can be multiple publishers and multiple subscribers. An example is shown in figure 2.7. The reader can consider a ROS topic as a bucket where ROS nodes can put or get information.
- *Service*: A ROS service provides a client-server paradigm for communication between ROS nodes. Both request and response are ROS messages. An example is shown in figure 2.8. The reader familiar with remote procedure calls (RPCs) can notice the similarity.
- *Master*: The ROS master is a special ROS node that offers registration and lookup services to all the other ROS nodes, letting them find each other to exchange ROS messages.
- *Naming*: ROS nodes, topics, and services are identified by unique names

in the computation graph. The naming system is hierarchical through namespaces. ROS allows remapping the names when a ROS node is launched, and this is a powerful mechanism that allows easy integration of different ROS nodes.

Besides the technical features, the greatest strength of ROS relies on the vast number of available packages developed and maintained by a large community. For example, the navigation stack [39] provides well-established algorithms that enable a robot to navigate and was used in this project.

2.4.2 V-REP

Virtual Robot Experimentation Platform (V-REP) [12, 13] is a versatile and scalable robot simulation framework that allows simulating quickly and precisely complex physical scenarios utilizing a physical engine. A simulation scene contains scene objects (e.g., shapes, vision sensors) arranged hierarchically in a tree structure. V-REP provides an extensive library of scene objects but also an integrated environment to create new ones.

Several programming paradigms are supported to execute control code. The most interesting ones for this work are:

- *Embedded scripts*: The code is written in Lua [40] scripts attached to scene objects, called child scripts. V-REP executes a simulation loop written in the main script, which calls the child scripts following the scene hierarchy. This paradigm is the most powerful because of its modular and distributed nature.

- *Remote API*: It is possible to control a V-REP scene from external code using the remote API. The external code, which is the client, can be written in several languages, among which Python, and can use the remote API functions almost as regular functions.

- *ROS interface*: V-REP can also launch a ROS node. In this case, child scripts can communicate with other ROS nodes by means of ROS topics and services, as described in section 2.4.1.

The previous work discussed in section 2.1 developed several scene objects, along with their child scripts using ROS topics and services. The most important one is the Turtlebot 2i with its sensors, described in section 2.4.6. Concerning this project, instead, the remote API was exploited to reset the scene from the RL environment.

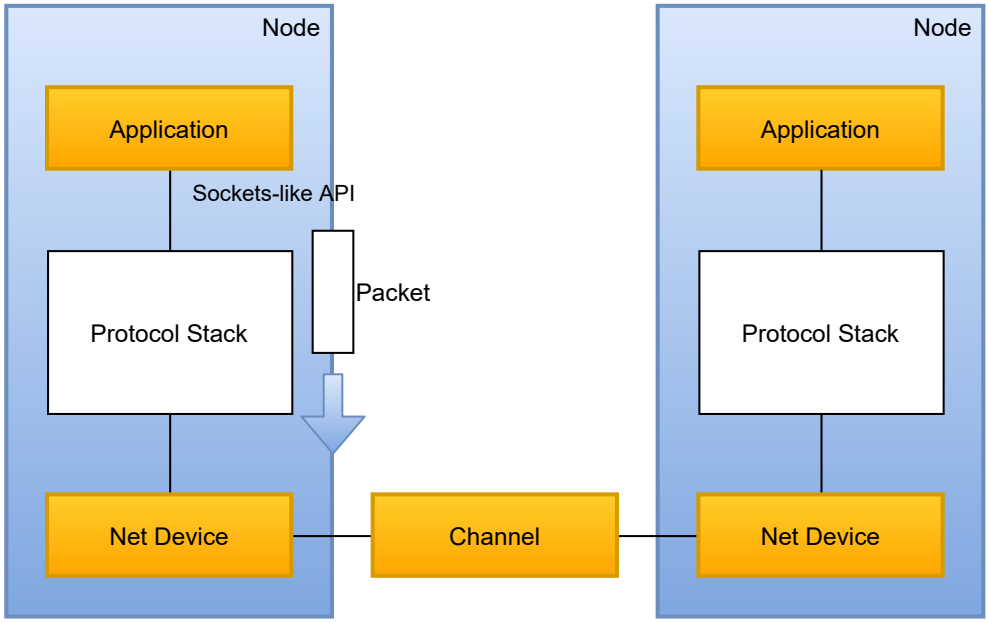


Figure 2.9: ns-3 key concepts.

Apart from its powerful functionalities, V-REP was preferred to other platforms (e.g., Gazebo [41, 42]) because it provides a library of scene objects for warehouses. Figure 2.1 shows an example of simulated warehouse in V-REP.

2.4.3 ns-3

Network Simulator 3 (ns-3) [14, 15] is an open-source network simulator and represents the de-facto standard for academic and industrial research in the telecommunication domain. It provides a large C++ library that includes today's most popular protocols and models for physical phenomena such as fading and interference.

The key abstractions used in ns-3, shown in figure 2.9 and also commonly used in networking, are [43]:

- *Node*: A ns-3 node represents a device connected to the network. For example, it might be a laptop, a smartphone, or a robot.
- *Application*: A ns-3 application is a user-level software running on a ns-3 node that performs a task using network services, which are accessed through a sockets-like API. An example is the *UdpEchoServerApplication* class, which implements an echo server.

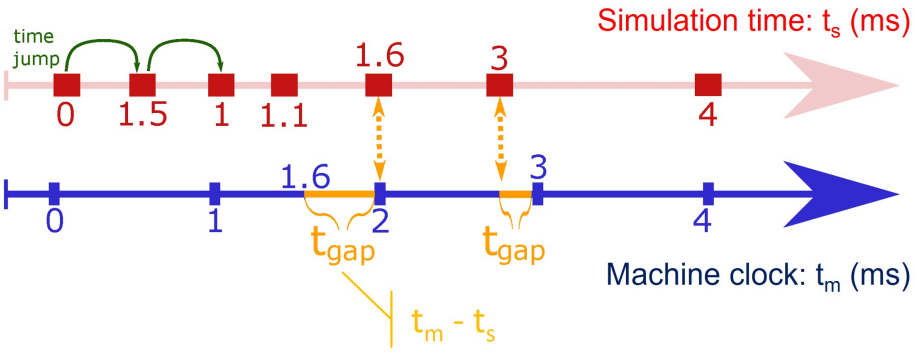


Figure 2.10: ns-3 real-time mode.

- *Channel*: A ns-3 channel abstracts the medium over which data flows. For instance, a wire which many nodes can access with Carrier-Sense Multiple Access (CSMA), such as Ethernet networks, can be represented with the *CsmaChannel* class.

- *Net device*: A ns-3 node can connect to a ns-3 channel only if a specific ns-3 net device is installed on it. In particular, a ns-3 net device abstracts the driver of a Network Interface Card (NIC). An example is the *CsmaNetDevice* class, which allows a ns-3 node to communicate over a *CsmaChannel*.

- *Packet*: A ns-3 packet encapsulates the data to transfer, known as payload, with the protocol information, known as the header, precisely like in real packet-switched networks.

ns-3 is a *discrete-event* simulator. All the simulated objects, such as ns-3 applications and net devices, schedule events to be executed at a specific simulation time. Thus, the simulator runs the scheduled events in sequential time order, jumping by default immediately to the next event when the current one is completed. This operating mode means that the simulation time is discrete, and there is no relationship between simulation time and real-time.

However, ns-3 supports also a real-time mode for integration with test-beds. In this case, the simulator attempts to keep the simulation clock aligned with the machine clock. Specifically, as shown in figure 2.10, when an event is finished, the simulator computes the gap between the machine clock t_m and the simulation time of the next event t_s: $t_{gap} = t_m - t_s$. If $t_{gap} < 0$, the next event is scheduled for the future and the simulator sleeps for t_{gap} before executing it. Otherwise, when $t_{gap} > 0$, the simulation is late because the

```
import gym

agent = get_agent()
env = gym.make('CartPole-v0')
env.seed(1)
for episode in range(20):
    env.render()
    state = env.reset()
    done = False
    while not done:
        action = agent.get_action(state)
        state, reward, done, _ = env.step(action)
env.close()
```

Algorithm 2.1: Simple usage of an OpenAI Gym environment.

computation of the previous event took too much time, so the next event is immediately executed. For heavy simulations, t_{gap} might increase indefinitely, leading to inaccurate simulations. For this reason, it is possible to change the synchronization mode from BestEffort (default) to HardLimit, which aborts the simulation if t_{gap} exceeds a user-defined threshold. In this project, the real-time mode was used for integration with V-REP and the *HardLimit* option was chosen to make sure the simulation is not too heavy and ns-3 keeps real-time.

2.4.4 OpenAI Gym

OpenAI Gym [44, 45] is a widely-used toolkit for defining and implementing RL environments in Python. In particular, it provides a collection of well-known problems, such as Atari games, and a library to simplify the implementation of new ones. The main idea is to decouple problem and agent so that different RL algorithms can be tested and compared, and this is realized by defining a simple API shared by all the environments. Algorithm 2.1 shows an example of usage of an OpenAI Gym environment through such an API, which consists of the following methods:

- *Step*: Applies the action to the environment and returns the new state and the reward, as well as a flag indicating if the episode has terminated.
- *Reset*: Resets the environment to an initial state and returns it. This method is called when an episode ends and a new one has to be started.

- *Render*: Renders the environment, for example, by showing it on display in a human-readable way. In order to save computation, it is possible to turn the rendering off.
- *Seed*: Sets the seed for the Random Number Generators (RNGs). This method enables the reproducibility of tests.
- *Close*: Performs the necessary cleanup before ending.

When implementing a new RL environment, it is worth doing it with OpenAI Gym since many libraries provide ready RL agents which assume this API. An example is Keras-RL [46], which was used in this project.

2.4.5 Keras-RL

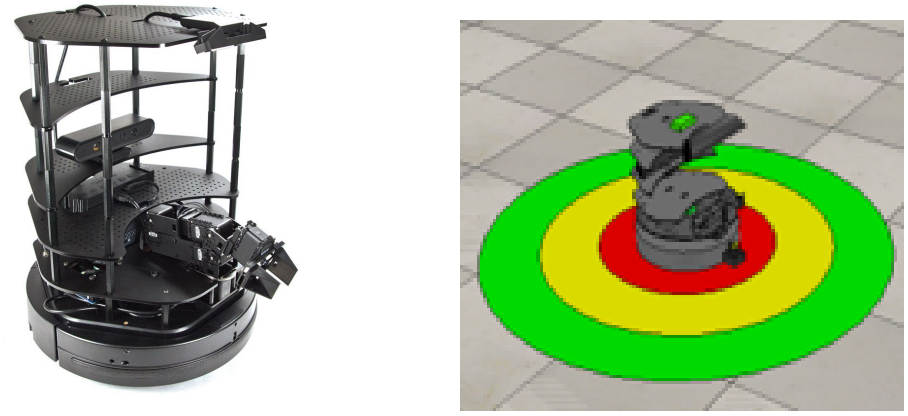
Keras-RL [46] provides a Python library of state-of-the-art DRL algorithms and works with OpenAI Gym environments. Behind the scenes, it uses the Keras API [47] for all the things related to DL, such as neural networks, optimizers, and loss functions.

Among other things, Keras-RL defines APIs for agents and callbacks through abstract classes. Therefore, it is possible to implement new agents and define new operations within the framework. In this project, Keras-RL was chosen mainly to get a ready implementation of DQN. On the other hand, some naive agents were implemented as Keras-RL agents to provide baselines, and logging was realized using Keras-RL callbacks.

2.4.6 Turtlebot 2i

Turtlebot 2i [21, 22] is a differential-drive mobile robot manufactured by Trossen Robotics. Like the other robots of the Turtlebot family, it is generally used for education and research purposes. The robot is shown in figure 2.11 and includes the following hardware:

- Mobile base: Kobuki Mobile Base
- Robotic arm: Pincher MK3 Robo Arm
- Sensors: Orbbec Astra Cam, Intel RealSense 3D Camera SR300-Series, Bumper Sensors, Scanse LiDAR
- Processor: Intel® Celeron® Processor J3455 (2 MB cache, up to 2.3 GHz)



(a) (b)

Figure 2.11: Real Turtlebot 2i (a) and V-REP model (b).

- Memory: 4 GB RAM DDR3L-1600MHz
- Storage: 240 GB SSD
- Connectivity: 802.11AC Wi-Fi, Bluetooth 4.0

ROS software is available for the mobile base, the robotic arm, and the sensors. The LiDAR is not included by default but was added for the previous work discussed in section 2.1. Turtlebot 2i was chosen because it is a low-cost mobile robot that can navigate and perform pick-and-place operations, thanks to the mobile base and the robotic arm.

2.5 Related work

Plenty of research work has been conducted to solve the resource management problem in MEC [11, 48, 49]. The key challenge is to decide for each device whether it should compute locally or offload. The main adopted approaches are game theory and DRL. Indeed, the former can naturally be applied by considering each MD as a player and results in efficient distributed algorithms [50]. At the same time, the latter can learn a policy dynamically and then use it to provide solutions efficiently [51]. Both approaches are used to find sub-optimal solutions to NP-hard optimization problems, which cannot be solved optimally in real-time.

Game theory was applied by Chen et al. [52] in a basic system with one MEC server and multiple MDs, each one with one task. In particular, the

authors optimized a linear combination of latency and energy, and they proved the solution to be a potential game, thus converging to a Nash equilibrium (NE) in a finite number of iterations. Other papers followed a very similar approach and extended [52]. Specifically, Yang et al. [53] also considered hard latency constraints of the tasks, load on the servers, and allowed MDs to offload also to MEC servers located in other 5G cells. Guo et al. [50] studied a three-layered architecture with cloud and edge, allowing MDs to choose whether to compute locally, offload to the MEC server, or offload to the cloud, and compared the game-theoretic solution to centralized brute force and greedy solutions. The same work was repeated also without cloud layer but with multiple tasks per MD [54]. Furthermore, [54] was used to build a dataset for training decision trees and to improve adaptability to changes in the scenario, such as varying number of MDs. Li et al. [55] also applied game theory in a specific scenario in which multiple robots have to perform 3D scene understanding. In that work, the authors exploited the knowledge about the particular task and allowed partial offloading. Indeed, since the scene understanding uses a CNN, a robot can also decide to forward the input until a certain layer of the CNN locally and to offload the rest, lessening the data sent on the network in comparison to complete offloading.

Concerning DRL, although not using the edge but only the cloud, a very relevant work was done by Chinchali et al. [56]. The authors optimized a linear combination of latency and energy with an Advantage Actor-Critic (A2C) agent located on the robot that does not need information from other devices, giving it a network budget[9] as a fair-share term. Since they focused on offloading ML tasks, in particular a face recognition task, they also proposed to use a less accurate model for local computation providing acceptable delays for onboard computation due to the fewer parameters and fewer operations. Furthermore, they introduced the novel idea of skipping the computation and reusing the last output, exploiting the temporal coherence[10] in video streams. Other papers jointly optimized task-offloading decisions and bandwidth allocation. Huang et al. [51] formulated the problem as mixed-integer non-linear programming, which is highly complex to solve, and then transformed it to a MDP in order to solve it faster with DQN. Another DRL solution was applied in [57] with the same goal, also adding a procedure that automatically adjusts the parameters of the algorithm on the fly.

All the discussed papers, except [56], model analytically the commu-

[9] The robot can offload at most B times for each T times, B being the network budget and T the finite horizon.

[10] Sequential frames are significantly correlated and can be very similar.

nication with a wireless interference model based on the Shannon-Hartley theorem [11], which allows estimating for each MD the throughput during the offloading operation. These works also perform numerical experiments using only synthesized data, in which ideal behavior and few phenomena are considered. However, in real implementations, such a communication model requires that the MDs candidate to offload measure their Signal-to-Interference-Noise Ratio (SINR) by sending constant pilot signals to the BS at the same time, as described in [52]. This mechanism is difficult to realize with common packet-based protocols such as TCP and UDP, so the procedure should be performed at low layers of the network protocol stack. In the case of game-theoretic solutions, this has also to be repeated until convergence to the NE. Moreover, these approaches imply a strong synchronization among the MDs, that have to wait until all the tasks are completed before repeating the offloading decision-making, possibly wasting time without using network resources. Another limitation of the reviewed papers, excluding [56], is given by the computation model. The capability of a device (MD or server) is abstracted with the CPU-cycle frequency f_{cpu}, whereas a task is represented by the pair (L, X), where L is the input-data size and X is the computation intensity (CPU cycles per bit). The execution time and energy are then computed with basic operations using these parameters. However, in real systems, this model is not very precise due to the varying load of the system (i.e., processes other than the considered task are always present) and the availability of GPUs beyond CPUs. Such issues make the solutions unsuitable and complex to apply in real scenarios, so they were not chosen for this degree project.

The solution developed in this degree project is mainly inspired by [56], which does not suffer from the limitations mentioned above as the DRL algorithm uses actual measurements of latency and energy in the reward without modeling communication and computation analytically. Moreover, since it does not rely on the wireless interference model to estimate the throughput, it works with any network technology (4G, 5G, and Wi-Fi) and any network topology (MEC servers in the BSs, APs, or in small-scale data centers). It can also be easily extended to include both edge and cloud layers. Another advantage is that other MDs using the network for operations other than MEC (e.g., human workers' smartphones accessing the Internet) are allowed. Each robot also acts independently, not synchronizing with the others, and this fits the strategy used in the previous work presented in section 2.1. Such features simplify the implementation and the deployment in real or simulated environments. Moreover, since the task to be offloaded in

this work is the scene understanding, both the less accurate model for local computation and the temporal coherence ideas fit perfectly.

It is worth highlighting that this work does not merely implement the solution in [56] for the HRC scenario, rather it develops several extensions, representing novel contributions. The most important contribution, novel in literature to the best of the authors' knowledge, is the usage of a safety KPI to achieve safety-oriented decision-making. Also, the solution is adapted for MEC instead of CC. Lastly, the network budget is replaced with actual network measurements. Indeed, such a hyperparameter should be set manually and would not be robust to dynamic changes in the scenario.

Chapter 3

Methods

This chapter provides the methods used to solve the task offloading problem for the scene understanding formulated in chapter 1. Section 3.1 describes the system design, including all the modules needed for the solution. Section 3.2, instead, presents the system implementation in a simulated environment using the frameworks and tools described in section 2.4.

3.1 System design

This work considers a MEC scenario with multiple robots and a single server. The MEC server is supposed to provide enough resources to all the robots through virtualization. So, despite the name used throughout the thesis, it is more realistically a small-scale data center.

The safety-oriented solution to the task-offloading problem for the scene understanding was designed as a system of several modules, some running on the robot and others on the MEC server. Figure 3.1 illustrates such a system. The red block represents the core decision-making logic, whereas the yellow ones are helper modules. The blue blocks, instead, are part of the safety framework of section 2.1, so they are not a direct contribution of this work. All these modules realize the decision-making for one robot, so there is one system for each robot.

The system aims to produce a scene graph continually and make it available for other processes on the robot (e.g., risk analysis and evaluation, risk mitigation). For doing this intelligently using MEC and considering safety, the task offloading module senses the environment state by cooperating with network monitor (for network aspects) as well as risk analysis and evaluation (for safety aspects). Thus, it decides whether to use the simplified scene

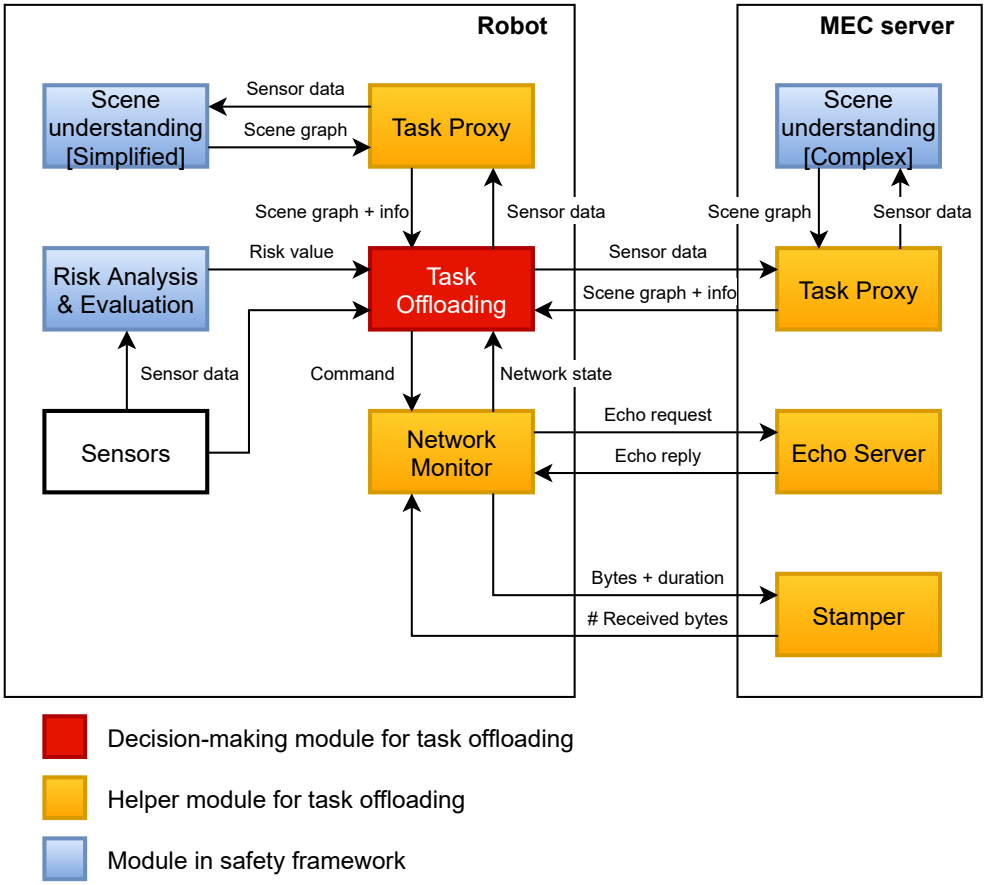


Figure 3.1: System design for task-offloading decision-making.

understanding available on the robot or the complex one on the MEC server. Another choice is to reuse the last scene graph, stored internally. When a scene graph is available on the task offloading module, the step terminates, and the system repeats the loop. The rest of this section describes in detail each component of the system.

3.1.1 Task Offloading

The task offloading module, highlighted in red in figure 3.1, is the core of the system and is responsible for deciding if the scene understanding should run on the robot or the MEC server. Since it uses RL, it provides the agent and its interaction with the environment. However, the latter is realized in cooperation with the other modules, which help sense the state, apply the action, and compute the reward function.

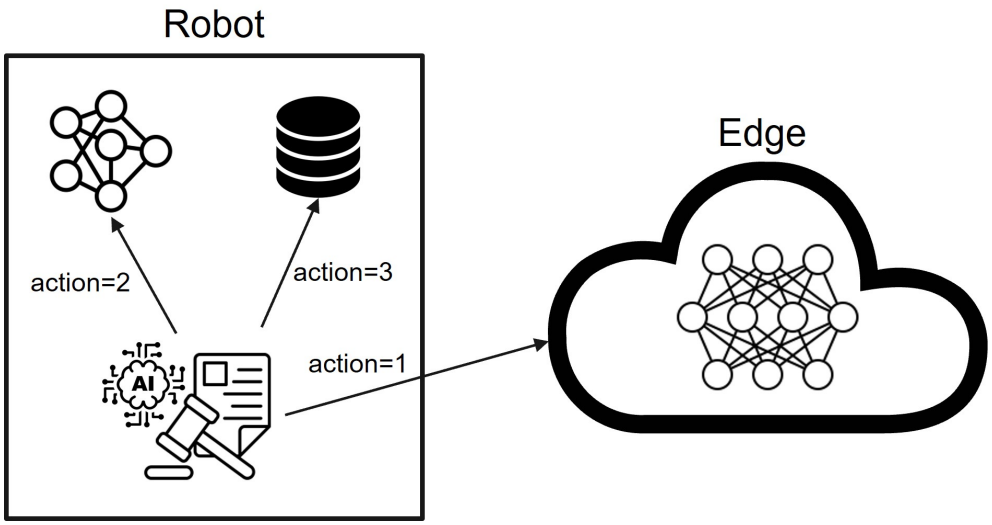


Figure 3.2: Action space in the RL safety-oriented task-offloading environment.

In line with section 2.3, the RL environment is formalized as an MDP. Therefore, state space S, action space A, and reward function R are defined below. The reader is reminded that the discount factor $\gamma \in [0, 1]$ is just a real number and does not need further definitions, whereas the transition function T is not available in RL.

Action space

Each action represents a way for the agent to perform the scene understanding. Therefore, each successful step terminates with a scene graph available on the task offloading module. The possible actions that the agent can take, illustrated in figure 3.2, are:

[A1] *Offload to the edge*: The robot sends the sensor data to the MEC server, which performs the scene understanding and returns the scene graph. On the MEC server, the scene understanding relies on a complex[11] CNN with high accuracy for instance segmentation.

[A2] *Compute on the robot*: The robot performs the scene understanding locally. In this case, the instance segmentation is done with a less complex CNN with lower accuracy than the one used on the MEC server.

[11] Complexity in CNNs is mainly measured by the number of parameters, thus it is related to the architecture (e.g., number of layers, number of filters per layer).

[A3] *Skip computation and reuse last output*: The robot does not perform any calculation, nor does it offload the task to the MEC server. It just reuses the previous output.

For action A1 some minor operations are also involved. Specifically, the RGB image is resized to I_{size}^{nw} before sending it on the network to speed up the transfer. I_{size}^{nw} can be the expected size of the CNN on the MEC server or smaller than that, with the latter case causing image degradation. The same resizing is applied to the depth image, given that it is combined with the RGB one and the output of the instance segmentation. In addition to this, the computation is aborted if the action takes more than the maximum allowed latency L_{max}. In such a case, the step is considered failed, and the scene graph is not available. The communication over the network uses TCP as transport layer protocol to have reliable delivery.

The key idea behind actions A1 and A2 is to provide a different trade-off between accuracy and complexity. Indeed, with the same hardware, the more complex a CNN is, the more operations and the longer its inference step takes. On the other hand, a CNN generally runs faster when a GPU is available on the machine. Therefore, this design uses a more complex CNN on the edge, where a GPU is available, and a less complex one on board, where only a low-cost CPU is present. In this setup, the local inference can satisfy the latency requirements of the risk management process. In addition to this, it is highlighted that, for these actions, the task offloading module only sends the robot's sensor data to the designated scene understanding module, which processes the input and returns the scene graph. In other words, the action is applied with the help of the scene understanding modules, and this can also be seen in figure 3.1.

Concerning action A3, it exploits the temporal coherence among sequential RGB images from the robot's camera. Indeed, when the current image is very similar to the last processed one, it is very likely that the new scene graph would be almost identical to the last one. Hence, it makes sense to skip the computation and reuse the last scene graph, which leads to immediate zero-latency output and energy savings.

State space

The environment state was designed to capture all the information useful for the agent to decide the action to take, in line with the reward function described later in this section. It includes several substates:

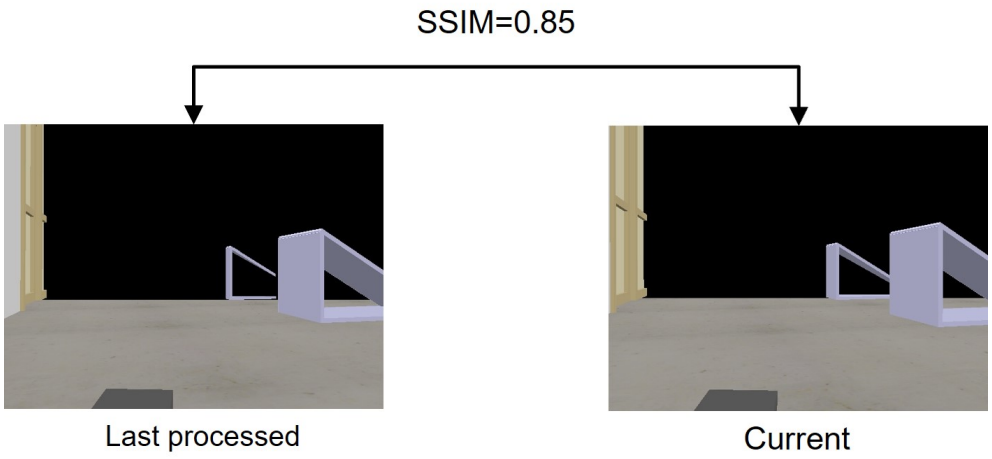


Figure 3.3: Temporal coherence measured with SSIM between two images captured within a short period of time.

[S1] *Network*: The network state from the robot's perspective is given by $throughput \in [0, S^{nw}_{max}]$ and $RTT \in [0, T^{ping}_{max}]$. Both attributes have continuous domains, S^{nw}_{max} represents the maximum network speed, whereas T^{ping}_{max} is the maximum allowed duration of the RTT measurement. The network monitor is responsible to sense and provide this substate on-demand.

[S2] *Safety*: The safety of the robot's surroundings is measured by the maximum risk value in the last output of the risk analysis and evaluation. Therefore, it can assume any real value in $[0, R_{max}]$, where R_{max} is the maximum risk value that the risk analysis and evaluation can output.

[S3] *Edge output*: The MEC server provides a more accurate scene understanding. This is represented with a Boolean value that is 1 if the last scene graph was computed on the edge, 0 otherwise.

[S4] *Temporal coherence*: The similarity between the current and the last processed images is measured with the Structural Similarity Index Measure (SSIM) [58]. An example is shown in figure 3.3. More specifically, this metric considers luminance, contrast, and structure of the images to compute a real value $SSIM \in [0, 1]$, where the boundaries 0 and 1 mean distinct and identical, respectively.

Substate S1 is the only one containing more than one feature. Therefore, the complete environment state can be obtained by unfolding it and counts five

attributes: throughput, RTT, risk value, edge output, and temporal coherence. The reason for including *edge output* (substate S3) is that it is useful in combination with the temporal coherence (substate S4) for action A3. For example, the agent might decide to skip the computation with a certain temporal coherence only if the last scene graph was computed on the edge, as a more accurate output is provided.

Since four of its five attributes are continuous, the environment state has a huge domain. Moreover, its evolution depends on complex dynamics (e.g., network phenomena, navigation of the robot) and external factors (e.g., the behavior of the human workers). Thus, it would be very complex, if not impossible, to model it analytically or to define static rules, and this justifies the need for an RL approach.

It is worth mentioning that the task offloading module senses the environment state with the help of other modules, as shown in figure 3.1. In particular, substate S1 is provided by the network monitor as will be described in section 3.1.2, whereas the risk analysis and evaluation described in section 2.1 is responsible for substate S2. Substates S3 and S4, instead, are calculated internally directly by this module.

Reward function

The reward function was designed to incorporate the objectives that the robot should learn. It provides positive rewards in the $[0, 1]$ interval. The reason for this is that negative rewards stimulate the agent to reach the goal as quickly as possible [59], but the RL task-offloading problem is infinite-horizon in nature, so there is no goal to reach. The pseudo-code is shown in algorithm 3.1. Input and output are self-explanatory, except for the latency that needs clarification. In this work, latency is the time elapsed from the agent's action decision to the scene graph received by the task offloading module. Therefore, in the case of edge computing, latency includes both execution and communication time.

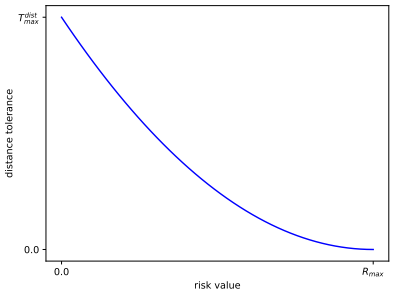
The reward function begins by checking whether the scene graph is available (line 3), meaning that the computation was not aborted. The reader is reminded that this can happen only for action A1, when the latency exceeds the limit L_{max}. If this check fails, the reward is set to 0, signaling that the agent's decision to offload was wrong.

Next, if the last scene graph was reused (action A3), it is compared with the current reference scene graph (lines 6 to 15) to check that: (i) all the objects in the current scene graph are present in the last scene graph, so no new object was missed (line 12); (ii) the difference in distance and direction for each

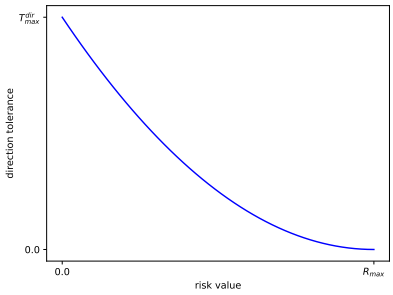
Algorithm 3.1: Reward function in the RL safety-oriented task-offloading environment.

Input: State s, Action a, Scene graph sg, Latency l
Output: Reward

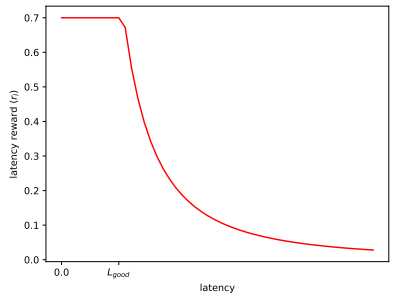
```
// Check success (step not aborted)
if sg = NULL then return 0

// Check scene graph
if a = A3 then
    sg_true ← ExtractSceneGraph()
    risk_value ← s["risk_value"]
    dist_tol ← GetDistanceTolerance(risk_value)
    dir_tol ← GetDirectionTolerance(risk_value)
    foreach obj_true in sg_true do
        if obj_true not in sg then return 0
        Δ_dist = |obj["distance"] − obj_true["distance"]|
        Δ_dir = |obj["direction"] − obj_true["direction"]|
        if Δ_dist > dist_tol or Δ_dir > dir_tol then return 0
    end
end

// Contributions to reward
r_l ← GetLatencyReward(l)
if a = A1 or (a = A3 and s["edge output"]) then r_a ← 0.3 else
  r_a ← 0
if a ≠ A1 then r_c ← 0.5 else r_c ← 0
if a = A3 then r_e ← 0.1 else r_e ← 0

// Safety weighs contributions
w_risk ← GetRiskWeight(risk_value)
return w_risk × (r_l + r_a) + (1 − w_risk) × (r_c + r_e)
```



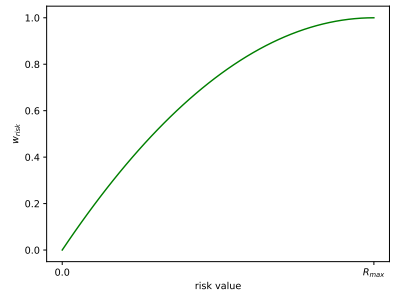
(a) $GetDistanceTolerance$



(b) $GetDirectionTolerance$



(c) $GetLatencyReward$



(d) $GetRiskWeight$

Figure 3.4: Functions used in algorithm 3.1.

object is within safety-based tolerances (lines 13 to 15). More specifically, the distance and direction tolerances are calculated using the following functions, also graphically shown in figures 3.4a and 3.4b, on the risk value:

$$GetDistanceTolerance\,(x) = T_{max}^{dist}\left(\frac{x}{R_{max}} - 1\right)^2 \tag{3.1}$$

$$GetDirectionTolerance\,(x) = T_{max}^{dir}\left(\frac{x}{R_{max}} - 1\right)^2 \tag{3.2}$$

where T_{max}^{dist} and T_{max}^{dir} are two hyperparameters indicating the maximum allowed variations in distance and direction, respectively, corresponding to risk value equal to 0. R_{max}, instead, is the maximum risk value that the risk analysis and evaluation can output. Keeping in mind that the input x is the risk value, the idea behind these equations is that the robot does not care about small inaccuracies in safe situations (e.g., a static object detected at $3.2\,\mathrm{m}$ instead of $3\,\mathrm{m}$ does not have a negative effect), whereas it is important to have an accurate and updated scene graph with mid-high risk (e.g., a small

variation in the direction of a human worker at $0.5\,\mathrm{m}$ makes it difficult for the robot to avoid them safely). In addition to this, as can be seen qualitatively in figures 3.4a and 3.4b, their trend is steeper than linear functions to consider medium risk values more as dangerous than safe. If the check fails, the reward function returns 0, indicating that the scene graph was inadequate for the safety requirements and unsuitable for reusing the last output.

If all the above-mentioned checks pass, the function continues by calculating several contributions for the final reward: a latency reward (r_l) inversely proportional to the latency (line 20); an accuracy reward (r_a) when the scene graph is output of the complex model on the MEC server (line 21); a congestion reward (r_c) when the network was not used (line 22); an energy reward (r_e) when the computation was skipped (line 23). In particular, the latency reward is calculated with the following function, also graphically represented in figure 3.4c, applied to the latency:

$$GetLatencyReward\,(x) = \begin{cases} 0.7 & \text{if } x \leq L_{good}, \\ 0.7\left(\frac{L_{good}}{x}\right)^2 & \text{if } x > L_{good}. \end{cases} \tag{3.3}$$

where L_{good} is a hyperparameter representing the latency achievable with little network congestion, thus it should consider the network technology. The latency reward saturates at such a value, avoiding large differences in rewards with small variations in latency. This function encodes the robot's desire to get latency under a certain threshold L_{good}.

The last part uses safety to calculate the final reward (lines 26 to 27). First, the risk value is mapped to a weight (w_{risk}, line 26) using the following function, also shown in figure 3.4d:

$$GetRiskWeight\,(x) = -\left(\frac{x}{R_{max}}\right)^2 + \frac{2x}{R_{max}} \tag{3.4}$$

where the same consideration made for equations (3.1) and (3.2) applies, meaning that medium risk values are treated more as hazardous than safe. After that, this weight is used to combine the contributions mentioned above (line 27). In particular, w_{risk} scales the contributions related to a safety improvement (r_l and r_a), whereas $1 - w_{risk}$ the others (r_c and r_e). The key insight is to give more importance to latency and accuracy with mid-high risk, whereas to encourage avoiding network congestion and saving energy with low risk. In other words, the robot is stimulated to adapt latency and accuracy to the safety requirements, using the edge only when the network performance

is good and when the situation is hazardous. On the other hand, when in safe conditions, the robot should not use the edge to let other robots in more critical situations use it.

Finally, a consideration has to be made for the $ExtractSceneGraph$ function (line 7) used to generate the reference scene graph in case of computation skip. Such a function cannot perform the scene understanding with instance segmentation, or the advantage of skipping the computation is lost. It needs to be fast. Section 3.2 provides information about its implementation in simulations, whereas chapter 5 gives insights on how to realize it in a real environment.

Agent

The RL environment defined above has a discrete action space and a continuous state space. Indeed, the former contains just three possibilities, whereas the latter has some continuous attributes (throughput, RTT, risk value, temporal coherence). Based on these features and on what described in section 2.3, DQN is a valid candidate. Considering also that it was previously applied successfully by Terra et al. for the risk mitigation module [8] and it was widely used in MEC [60, 61, 57, 62], it was chosen for the solution.

Other alternatives exist and could be used without any modifications of the RL environment. For example, Cross-Entropy Method (CEM) [63] and State–Action–Reward–State–Action (SARSA) [28] also work with discrete action space and continuous state space. However, these agents are out of the scope of this work and are left for possible future work.

3.1.2 Network Monitor

The network monitor provides a service for measuring on-demand the end-to-end network performance between robot and MEC server. In other words, since the module runs on board, it enables the robot to sense the network state, corresponding to substate S1 of the RL environment state. As already mentioned, the two metrics used to abstract the network performance are RTT and throughput. The measurement, detailed in the following, is realized in cooperation with two modules installed on the MEC server, namely the echo server and the stamper, as shown in figure 3.1.

After receiving the command, the network monitor starts with a ping operation. Specifically, it sends an ICMP echo request to the echo server and waits for the ICMP echo reply. The echo server is responsible for sending back such a reply and is part of the kernel in popular OSs. In this way, the network

monitor can calculate the RTT, which is the time elapsed from the request to the receipt of the reply:

$$RTT = t_{reply} - t_{request} \tag{3.5}$$

where $t_{request}$ is the time the request is sent, t_{reply} is the time the response is received, and all the quantities are expressed in ms. As usual with ping operations, the communication between network monitor and echo server uses UDP at the transport layer, which is connection-less and so it can immediately send packets without overhead, unlike TCP which has an initial handshake. In case the operation takes more the maximum allowed duration T_{max}^{ping}, the network monitor sets $RTT = T_{max}^{ping}$.

Once the RTT measurement is concluded, the network monitor estimates the throughput from robot to MEC server by sending bytes to the stamper for a duration of T_{tp}. The stamper waits for T_{tp} and then sends back the number of received bytes n_{recv}. Therefore, the network monitor can calculate the estimated throughput as follows:

$$throughput = \frac{n_{recv}}{T_{max}^{tp}} \cdot 8 \cdot 10^{-6} \tag{3.6}$$

where the multiplication factor converts the result in Mbps, as common when dealing with network speeds. Such a communication uses TCP at the transport layer, as the task offloading module does when sending the inputs to the scene understanding. In this way, the throughput estimate is more significant.

It is worth noticing that, since one of the objectives of the whole system is to reduce the latency, the measurement needs to be fast. The network monitor guarantees this requirement, since the duration has an upper bound of $T_{max}^{ping} + T_{tp}$. Moreover, the measurement is independent of the topology, so it fits any position of the MEC server (in the BS, AP, or small-scale data centers) and also CC, even though the latter is not considered.

3.1.3 Task Proxy

A task proxy acts between a scene understanding and the task offloading module and cooperates with them when performing an action. Specifically, the goal is to calculate the communication latency l_{comm} and the execution latency l_{exec}, which are used by the task offloading module to estimate the energy consumption, which is of interest in the experiments.

When the task offloading module takes an action, it sends the sensor data

to the task proxy, which forwards it to the scene understanding and waits for the output. Thus, the task proxy can calculate the execution latency as:

$$l_{exec} = t_{end}^{exec} - t_{start}^{exec} \tag{3.7}$$

where t_{start}^{exec} is the time the sensor data is forwarded and t_{end}^{exec} is the time the scene graph is received from the scene understanding.

Next, the task proxy sends back the scene graph together with l_{exec} to the task offloading module. The latter can therefore calculate the latency:

$$l = t_{end}^{action} - t_{start}^{action} \tag{3.8}$$

where t_{start}^{action} and t_{end}^{action} are start and end time of the whole procedure. By difference, the task offloading module can also find the communication latency:

$$l_{comm} = l - l_{exec} \tag{3.9}$$

which is due to the network in case of edge computing (action A1) whereas negligible[12] for local computing (action A2).

It is worth highlighting that the task proxy for edge computing (action A1) runs on the MEC server to understand when the communication ends and the execution starts. Even though local computing (action A2) does not use the network, a task proxy is still used for symmetry to simplify the implementation. In addition to this, the implementation in the simulated environment gives more responsibilities to the task proxy, as described in section 3.2.

3.2 Implementation

The implementation was done in a simulated environment, with V-REP and ns-3 realizing the physical objects and the network, respectively. All the modules in figure 3.1 run on the same machine and are implemented using ROS, which both provides them communication and also integrates V-REP and ns-3. The implementation is shown in figure 3.5. Many of the ROS nodes are self-explanatory since they implement the homonyms[13] of figure 3.1. Therefore, the rest of this section focuses only on those deserving particular attention.

[12] There is still a small delay due to inter-process communication (IPC), i.e., communication between processes through the kernel.

[13] Risk analysis and evaluation was abbreviated to *risk_assessment*.

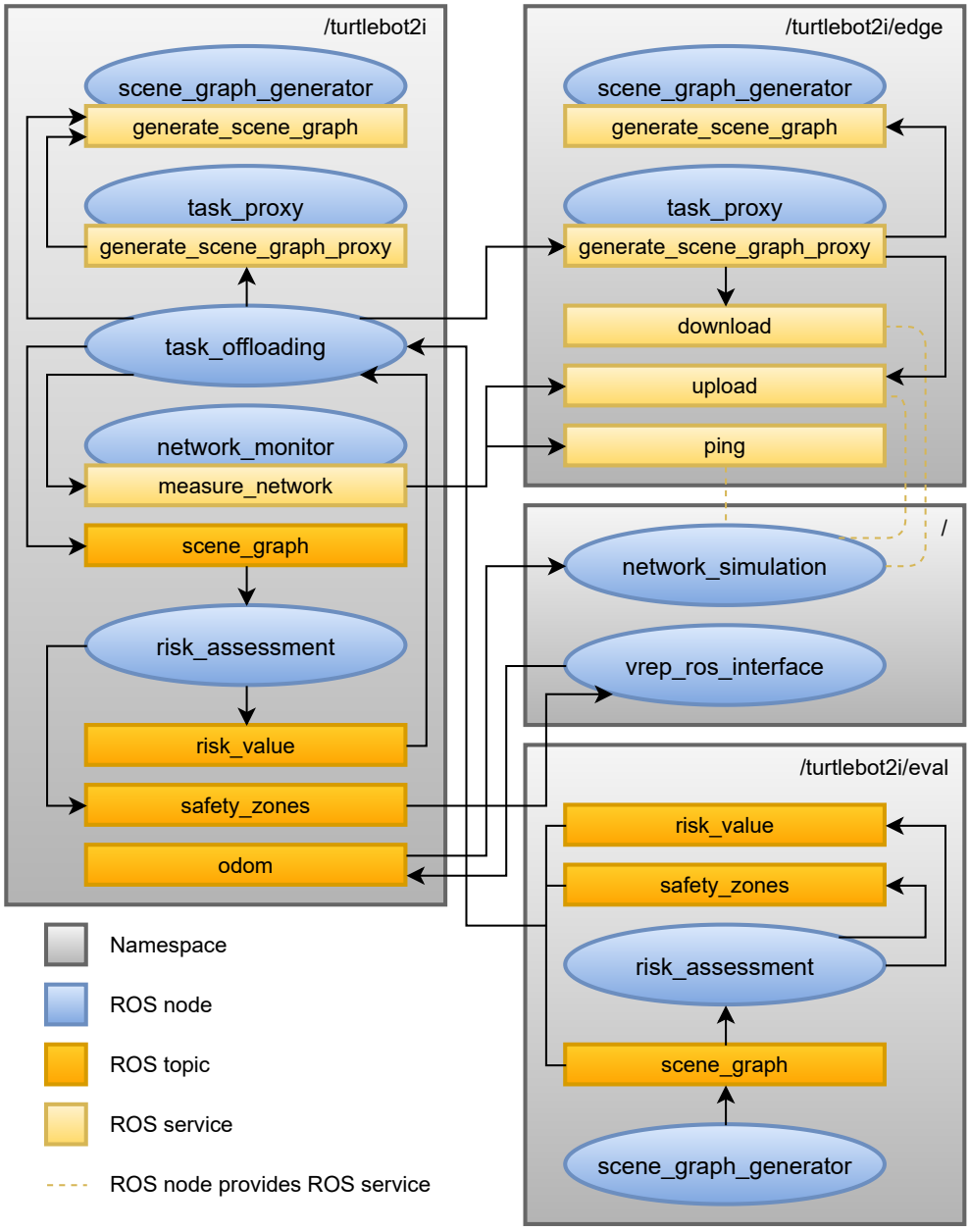


Figure 3.5: Implementation of system, including additional nodes for the evaluation, with ROS.

It is worth mentioning that, even though figure 3.5 shows the implementation for one robot, multiple robots can be used just by launching everything inside the */turtlebot2i* namespace (including the nested namespaces) under another namespace (e.g., */turtlebot2i_2*). Moreover, even though the warehouse scenario, the network topology, and the DQN architecture are relevant contributions and part of the work, they are presented in chapter 4 as related to the setup of the experiments.

3.2.1 System

In the simulated environment, the network is realized with ns-3 but has to be used on-demand by robots in V-REP. This is realized by the *network_simulation* ROS node that:

- Simulates the network using ns-3 in real-time mode with the *HardLimit* option. This is done in a dedicated thread, so that the process can also do what described in the next bullet points. The simulated network contains ns-3 nodes for each robot in V-REP and for the MEC server. The stamper and echo server of figure 3.1 are implemented by installing a *PacketSink* and a *UdpEchoServer* ns-3 application on the ns-3 node for the MEC server, respectively.

- Provides ROS services for using the network. In particular, the *upload* and *download* ROS services transfer bytes from robot to MEC server and vice versa, respectively, while the *ping* ROS service realizes the ping operation needed by the network monitor. All of them basically schedule events in ns-3, which runs and executes them concurrently in a separated thread.

- Keeps synchronized the positions of robots in V-REP and corresponding ns-3 nodes. This is done by subscribing to the *odom* ROS topic, on which V-REP publishes the robot's position, and by updating the position of the ns-3 node.

Such a multi-threaded strategy integrates ns-3 and V-REP using ROS and is possible because ns-3 is thread-safe in real-time mode. To the best of the authors' knowledge, this is a novel approach not present in literature.

Another problem in the simulated environment is related to the execution of the complex scene understanding on the MEC server. Indeed, since commercial PCs like the one used for this project are usually not provided with the powerful GPUs available on edge or cloud, the simulated edge

computation would take longer than the real one. This is solved by the *scene_graph_generator* ROS nodes, which extract the scene graph directly from V-REP by querying it via remote API. Such an operation is much faster than the real scene understanding and, combined with the modified task proxy described in the following, allows pretending to perform the complex scene understanding with a powerful GPU even without having it. In addition to this, it is also ideal to implement the $ExtractSceneGraph$ function in algorithm 3.1 (line 7). However, the extracted scene graph is always perfect, so the different accuracy between onboard and on-edge instance segmentation models is not simulated, and this is one of the limitations discussed in section 1.6. This ROS node was already available from previous work. Still, it was extended to be alternatively either on-demand with a ROS service or continuous by publishing to a ROS topic. The system uses the former mode, but the latter is exploited for the evaluation, as described in section 3.2.2.

The *task_proxy* ROS node implements the homonym in figure 3.1 by providing the *generate_scene_graph_proxy* ROS service, but with the extended responsibility of simulating communication and execution latencies. In particular, the former is done only for edge computing (action A1) by using the *upload* ROS service to upload the sensor data from robot to MEC server and the *download* ROS service to download the scene graph from MEC server to robot. The latter, instead, is simulated by sleeping for:

$$t_{sleep} = L_{exec} - (t_{end}^{exec} - t_{start}^{exec}) + \sigma \tag{3.10}$$

where $L_{exec} = L_{exec}^{edge}$ or $L_{exec} = L_{exec}^{robot}$ are hyperparameters for edge and local computing, respectively, and $\sigma \sim N(0, 0.1 \cdot L_{exec})$ adds a Gaussian noise. The execution latency then becomes:

$$l_{exec} = L_{exec} + \sigma \tag{3.11}$$

The RL environment is implemented in the *task_offloading* ROS node using OpenAI Gym. It is highlighted that it requests the scene understanding by calling the ROS services provided by the *task_proxy* ROS nodes, but then it publishes the resulting scene graph on the *scene_graph* ROS topic to make it available to all the other ROS nodes (e.g., *risk_assessment*). In addition, it also bypasses the *task_proxy* ROS node and calls directly the ROS service provided by the *scene_graph_generator* ROS node for the $ExtractSceneGraph$ function in algorithm 3.1 (line 7). Moreover, although not shown in figure 3.5, it provides a pick-and-place navigation between randomly sampled shelves and conveyor belts in the warehouse by sending

navigation goals to the *move_base* ROS node. Since for experiments it is necessary to define an end also for infinite-horizon problems, the episode length E_{len} is measured in completed pick-and-place operations. Besides ROS, the RL environment also uses the V-REP remote API to reset the environment.

The RL agent is also contained in the *task_offloading* ROS node. DQN is provided by the Keras-RL library, which works with OpenAI Gym environments. Since neural networks learn better with normalized inputs, the state is pre-processed by means of min-max normalization. In particular, RTT, throughput, and risk value are normalized using T^{ping}_{max}, S^{nw}_{max}, R_{max}, respectively, which are hyperparameters representing the maximum values (the minimum values are 0).

3.2.2 Evaluation

The evaluation of the solution discussed in chapter 4 is realized in two steps:

1. During the simulations, some loggers record all the necessary information. Such loggers are implemented as Keras-RL callbacks so that they can be attached to the Keras-RL agent. Thus, they run inside the *task_offloading* ROS node.

2. After the simulations, the logs are analyzed with a specific Python script which calculates the metrics and generates charts.

Importantly, the logs related to safety come from the *risk_assessment* and *scene_graph_generator* ROS nodes. However, in order for this evaluation to be fair, such information should be available with low latency independently of the agent, and this is realized in the */turtlebot2i/eval* namespace. In this case, the *scene_graph_generator* ROS node runs in continuous mode and publishes at a constant rate the scene graph to the ROS topic.

Chapter 4

Results and analysis

This chapter describes the experiments carried out to evaluate the solution and discusses their results. Section 4.1 explains the experimental setup with all the settings and the reasons behind them. Section 4.2 presents the evaluation framework, including baselines and metrics. Finally, section 4.3 reports and analyzes the results.

4.1 Experiments

In order to evaluate systematically the solution designed in chapter 3, two RL agents were considered:

- DQN-2: The DQN agent can choose between local and edge computing (actions A1 and A2), but cannot skip the computation (action A3).
- DQN-3: The DQN agent can choose among actions A1 to A3.

DQN-2 helps to understand whether the RL algorithm manages to learn well when to offload the scene understanding, without action A3 which can affect such an analysis. DQN-3, instead, represents the complete solution and, by comparing with DQN-2, shows if skipping the computation is beneficial. This sequence was also followed to tune the hyperparameters used in the reward function. Furthermore, some naive agents were also tested and used as baselines, as presented in section 4.2.

The simulated scenario, described and motivated in section 4.1.1, contains up to two robots. From that, the following steps for the experiments were carried out:

1. Train DQN-2 and DQN-3 in the scenario with one robot.

2. Test DQN-2, DQN-3, and the baselines in the scenario with one robot.
3. Test DQN-2, DQN-3, and the baselines in the scenario with two robots, where both robots use the same agent.

The last step is useful to understand if the solution works with multiple robots offloading the scene understanding, effectively sharing the network resources. Moreover, it tests the generalization of DQN to slightly different states, as the new robot introduces different levels of network congestion.

Since the problem is infinite-horizon and the end of an episode is artificially defined as described in section 3.2, one episode can be considered as an entire run. Therefore, the tests were run for 10 episodes, and then the mean and standard deviation of the results were calculated. The rest of this section describes the simulated scenario and lists the settings for all the hyperparameters.

4.1.1 Simulated scenario

The simulated scenario was designed with the main idea of letting the robots experience different degrees of risk and a wide variety of network conditions. It includes two interconnected aspects: warehouse and network topology. The former includes all the objects physically present inside the warehouse and is simulated in V-REP. The latter, instead, indicates the arrangement of the devices in the network and is simulated in ns-3.

The warehouse, shown in figure 4.1, contains many static and dynamic objects that enable the robots to run into many situations with different risk values. The static objects are walls, pillars, boxes, conveyor belts, and shelves. Among these, only the walls are known a priori by the robots[14]. As for the dynamic objects, instead, there are human workers and robots. The former move according to a built-in AI provided by V-REP, whereas the latter perform the pick-and-place navigation described in section 3.2. For guaranteeing smooth navigation in the whole warehouse, one conveyor belt is placed in each room of the lowest row, whereas there is one shelf per room in the rest. Moreover, the simulation of robots demands more computing power than static objects and human workers because of the sensors and all the external ROS nodes. For this reason, in order to keep a frame rate of at least 10 FPS in V-REP, the scenario contains only up to two robots.

As a building, the warehouse is 30 m wide and 30 m long. Such a size provides enough space for the robots to meet varying distances from the AP,

[14] This means that they are in the global cost map of the ROS navigation stack

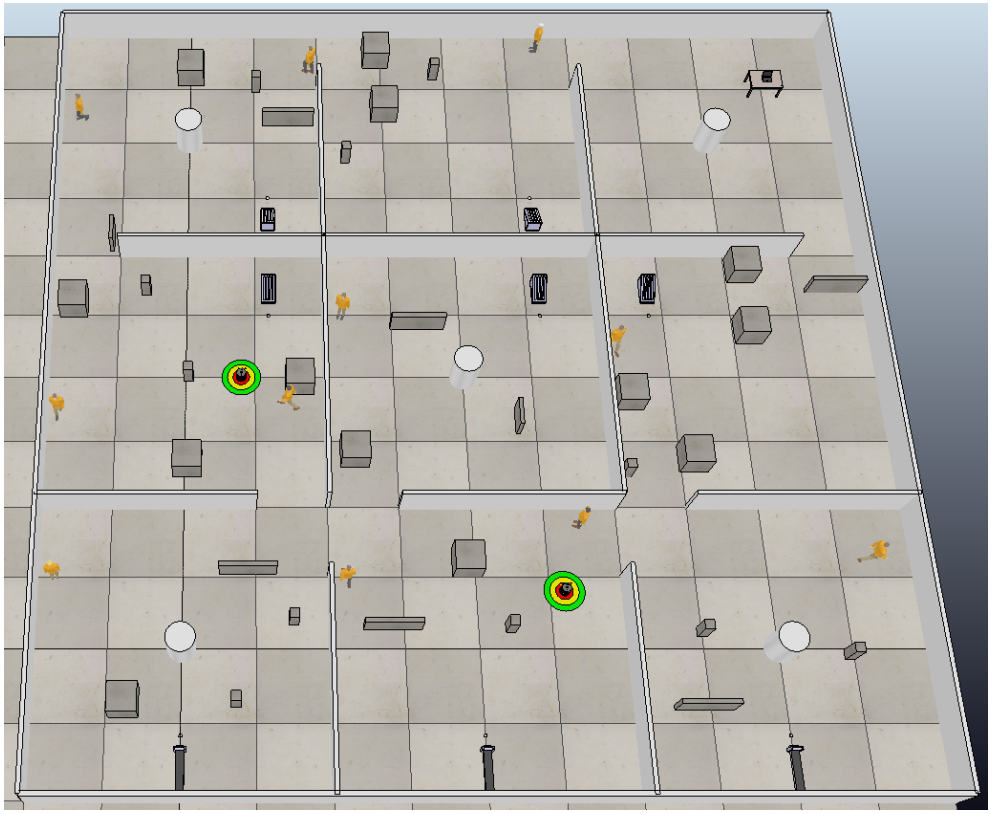


Figure 4.1: Warehouse scenario used for the experiments, simulated with V-REP.

which is correlated to the throughput. The space is divided in a 3×3 grid of equal-size square rooms. In this way, the walls are also considered in the network simulation by using the ns-3 *Building* class, which allows only square rooms, so the fading phenomenon is simulated.

Figure 4.2 illustrates the network topology, overlapped to the warehouse to understand the position of the ns-3 nodes. It is very simple: a Wi-Fi 802.11g LAN provides wireless connectivity in the entire warehouse and a MEC server is directly connected with a point-to-point link to the AP. The 802.11g standard was chosen over more powerful technologies because it can be congested with less traffic, helping ns-3 to keep real-time with the *HardLimit* option, as described in section 2.4[15]. The point-to-point link, instead, provides $100\,\mathrm{Mbps}$, which is by far more than what Wi-Fi 802.11g can achieve and therefore it is not the bottleneck of the topology, and has a delay

[15] Wi-Fi 802.11n (high throughput) with larger traffic was tried but resulted in ns-3 not being able to keep real-time with the *HardLimit* option.

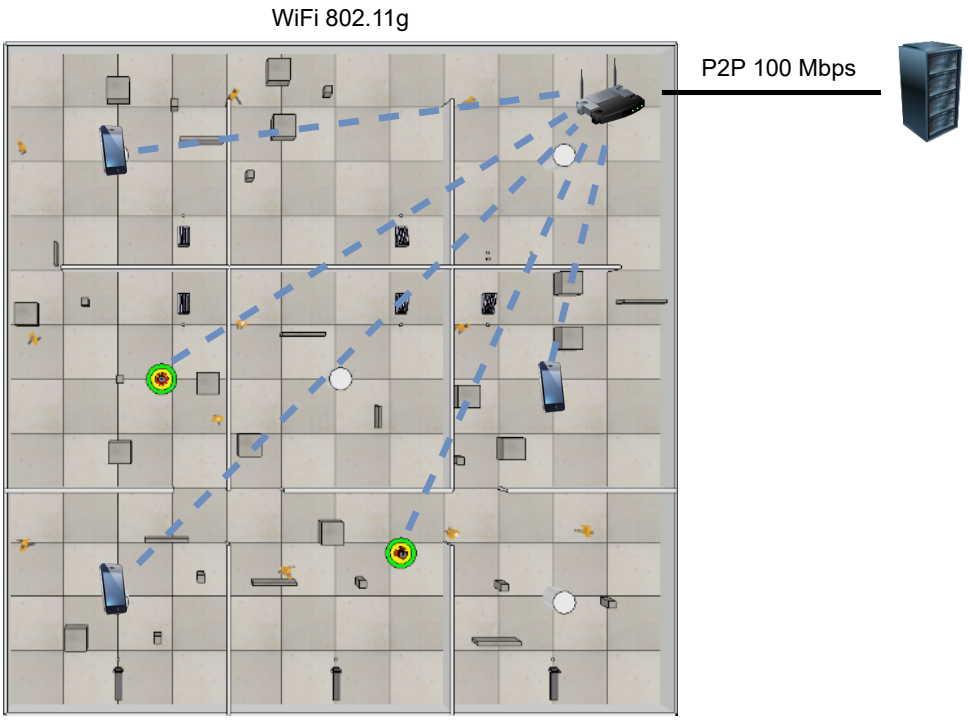


Figure 4.2: Network topology used for the experiments, simulated in ns-3. It is overlapped to the warehouse to clarify the position of the ns-3 nodes. The smartphone icons represent congesting nodes.

of $2\,\mathrm{ms}$, corresponding roughly to a distance of $400\,\mathrm{km}$. The AP is positioned in the upper-right corner so that the robots can experience a wide range of distances from it.

In the wireless LAN there are two types of ns-3 nodes: robots and congesting nodes. The robots use the network to communicate with the MEC server when offloading the scene understanding. The congesting nodes, represented with smartphone icons in figure 4.2, are instead virtual devices simulated only in ns-3 that use the edge randomly. More specifically, they are ns-3 nodes on which an *OnOffApplication* ns-3 application is installed. Such an application dynamically switches between the on and off states: in the on state it generates CBR traffic at B_{cong} rate towards the MEC server, while in the off state, it does not use the network. The duration of the on state (t_{on}) and the duration of the off state (t_{off}) are uniform random variables:

$$t_{on} \sim U\left(T_{on}^{min}, T_{on}^{max}\right) \tag{4.1}$$

$$t_{off} \sim U\left(T_{off}^{min}, T_{off}^{max}\right) \tag{4.2}$$

Table 4.1: Settings used in the experiments for the hyperparameters of the system.

HP	Value	Brief description
L_{exec}^{edge}	$0.195\,\mathrm{s}$	Execution latency on MEC server.
L_{exec}^{robot}	$1.05\,\mathrm{s}$	Execution latency on robot.
L_{max}	$2\,\mathrm{s}$	Max. allowed latency before aborting.
L_{good}	$0.4\,\mathrm{s}$	Saturation value for latency reward.
T_{max}^{dist}	$0.5\,\mathrm{m}$	Max. distance tolerance (reward function).
T_{max}^{dir}	$30°$	Max. direction tolerance (reward function).
T_{max}^{ping}	$100\,\mathrm{ms}$	Max. duration of RTT measurement.
T_{tp}	$0.1\,\mathrm{s}$	Duration of throughput measurement.
P_{cpu}	$10\,\mathrm{W}$	Power consumption of robot's CPU.
P_{nic}	$0.1\,\mathrm{W}$	Power consumption of robot's NIC.
S_{max}^{nw}	$24\,\mathrm{Mbps}$	Max. network speed.
I_{size}^{nw}	224×224	Size of RGB and depth images over the network.
T_{on}^{min}	$2\,\mathrm{s}$	Min. time in on state (congesting node).
T_{on}^{max}	$5\,\mathrm{s}$	Max. time in on state (congesting node).
T_{off}^{max}	$5\,\mathrm{s}$	Min. time in off state (congesting node).
T_{off}^{min}	$10\,\mathrm{s}$	Max. time in off state (congesting node).
B_{cong}	$3\,\mathrm{Mbps}$	Data rate in on state (congestion node).
R_{max}	4	Max. risk value.
E_{len}	5	Pick-and-place operations per episode.
γ	0.99	Discount factor for expected return in RL.

where T_{on}^{min}, T_{on}^{max}, T_{off}^{min}, T_{off}^{max} are hyperparameters. Therefore, the number of active (i.e., in the on state) congesting nodes varies dynamically and this provides different degrees of network congestion, which would not be possible to achieve with only two robots. The scenario contains exactly 3 stationary congesting nodes at the positions indicated in figure 4.2. This number was chosen experimentally as the highest value allowing ns-3 to keep real-time.

4.1.2 Settings

The values used in the experiments for all the hyperparameters encountered in this thesis are reported in table 4.1. Some motivations for these values as well as further settings for DQN and ns-3 are discussed below.

Regarding the execution, a Mask R-CNN with ResNet101 as backbone was considered for edge computing, in line with the previous work described

in section 2.1, and the MEC server was supposed to be provided with an Nvidia Tesla M40 GPU. Thus, L_{exec}^{edge} was set according to the results in [24]. For local computing, instead, YOLACT [64] with ResNet50 as backbone was considered, which is, to the best of the authors' knowledge, the fastest among the state-of-the-art models for real-time instance segmentation [65, 66]. Unfortunately, there is no measurement of its performance on Turtlebot 2i's CPU in literature[16], so L_{exec}^{robot} was estimated in the following way. First, YOLACT was optimized for Intel CPUs by using OpenVINO [67]. Then, it was run on the CPU available on the PC used for this project, calculating the mean inference time. Finally, this value was converted to an estimate for the robot's CPU by using benchmarks [68]. Such execution latencies consider only the instance segmentation in the scene understanding, as the other operations to build the scene graph are negligible in terms of time. As for the power consumption, P_{cpu} was set to the power consumption of Turtlebot 2i's CPU [69], whereas P_{nic} to that of common NICs for MDs [70].

Concerning the network, I_{size}^{nw} was set to a value very widely used in CV. Then, L_{good}, T_{max}^{ping}, and T_{tp} were adjusted accordingly. In particular, the maximum duration of the network measurement $T_{max}^{ping} + T_{tp}$ is half of the good latency L_{good} to guarantee a good estimate of the throughput without impacting too much on the system. In fact, with a too-small T_{tp}, such a measurement is very inaccurate, for example, due to the overhead of the three-way handshake in TCP. S_{max}^{nw}, instead, was measured experimentally in the simulation but, since it is used only for the min-max normalization of the throughput, a small error in such a value is not important. As for the network congestion, the reason why $T_{off}^{min} > T_{on}^{min}$ and $T_{off}^{max} > T_{on}^{max}$ is to increase the probability of having all the congesting nodes in the off state so that the robots can experience more frequently an excellent network state. In addition, B_{cong} was tuned experimentally to provide throughput close to 0 when all the congesting nodes are in the on state.

Some settings were conditioned by the implementation of the safety framework available from previous work. In particular, R_{max} was set to the maximum value that can be output by the implementation of the risk analysis and evaluation. Moreover, as mentioned in section 2.1, there are two implementations for the risk mitigation available from previous work. For the experiments, the FL-based one was chosen because it does not require training.

With regards to DQN, Keras-RL was used for the RL agent and as a

[16] A direct measurement on the robot, available physically at Ericsson's headquarters in Kista, was not possible either, because the work was carried out at home due to the COVID-19 pandemic, as mentioned in chapter 1.

consequence the DL part (e.g., DNN, optimizer) was provided by using Keras. The DNN was built with 3 fully connected hidden layers of 16 neurons each and ReLU activation [71]. The length of the training was set to 10^5 steps, with the agent using the ϵ-greedy policy with $\epsilon = 0.1$ and updating its target network every 10^4 steps. Adam [72] was chosen as optimizer, with a learning rate of 10^{-3}. Instead, in the test phase, the agent was configured to use a greedy policy, which always takes the action with the highest Q-value. These settings resulted in a convergence of DQN, monitored through mean Q-value and loss.

In ns-3 a couple of settings were adjusted. Since it does not support the automatic MSS tuning based on MTU discovery, which is instead done by real routers, the MSS was increased to 1472, and this resulted in a much more performing network compared to the default value. Such a value was taken from an example available in the documentation of ns-3 [73]. Moreover, the YANS model [74] was used for Wi-Fi channel and physical layer because, according to the ns-3 documentation, it is the most suitable choice for Wi-Fi-only channels (i.e., no mixed technologies on the same channel).

As a final remark, the focus is not on finding optimal settings, rather on proving the soundness of the solution. Therefore, some settings were not profoundly tuned. For instance, Adam was chosen as the optimizer without trying alternatives. In addition to this, all the configurations not explicitly mentioned in this section were left to the default values.

4.2 Evaluation framework

For evaluating the RL agents in a quantifiable way, it is necessary to design an evaluation framework made of baselines and metrics. The rest of this section presents it.

4.2.1 Baselines

The following baselines were used:

1. *Edge-only policy*: The agent always takes action A1.
2. *Robot-only policy*: The agent always takes action A2.
3. *Random sampling*: The agent randomly chooses between actions A1 and A2.

4. *Without risk mitigation*: The risk mitigation in the safety framework of section 2.1 is disabled, so the risk management process does not affect the navigation.

Baselines 1 to 3 are naive agents with no intelligence and are inspired by related work in MEC [56]. The edge-only policy (baseline 1) also corresponds exactly to the distributed architecture illustrated in figure 1.2. These baselines are useful to assess the offloading decision-making of the RL agents. They were implemented as Keras-RL agents, so that they can easily replace DQN.

Baseline 4, instead, was previously used in [8] and is helpful to assess the improvements in terms of safety provided by the safety framework, which is indirectly affected by the task offloading agent. The offloading agent is not relevant for this baseline, as the interest is only in observing how safe the navigation is. It is worth mentioning that the robot can avoid obstacles also without the safety framework, with the use of sensors such as LiDAR. However, in this way, the robot does not know semantic information about the objects (e.g., object type), so it cannot adapt the navigation to the different safety requirements.

4.2.2 Metrics

The metrics used for the evaluation can be classified into two groups: offloading and safety metrics. Before listing them, the reader is reminded that a step in the RL safety-oriented task-offloading environment is failed if the computation is aborted due to latency exceeding the limit L_{max}. Moreover, the latency is calculated as in equation (3.8), whereas the energy consumption is:

$$e = \begin{cases} P_{nic} \cdot l_{comm} & \text{if } a = 1, \\ P_{cpu} \cdot l_{exec} & \text{if } a = 2, \\ 0 & \text{if } a = 3 \end{cases} \tag{4.3}$$

where a is the action, P_{cpu} and P_{nic} are hyperparameters representing the power consumption of CPU and NIC, respectively, l_{exec} is the execution latency from equation (3.7), and l_{comm} is the communication latency from equation (3.9).

The offloading metrics evaluate the task offloading and are a direct contribution of this work, as they were selected by analyzing the related work in MEC and by reasoning about the specific problem addressed in this project. Specifically, the list is as follows:

- *Latency*: The mean latency per step, considering all the steps.

- *Latency no failures*: The mean latency per step, considering only the successful steps.
- *Latency per output*: The mean latency per output. It is calculated as the sum of latencies considering all the steps divided by the number of successful steps.
- *Risk* $\times$ *latency*: The mean risk value multiplied by latency, considering all the steps.
- *Success*: The percentage of successful steps.
- *Edge output*: The percentage of steps with scene graph output of the accurate model on the MEC server.
- *Valid scene graph*: The percentage of steps with scene graph passing the checks in the reward function at lines 6 to 15 of algorithm 3.1 (i.e., no new object missed and tolerance in distance and direction of each object satisfied), considering only the successful steps.
- *Energy*: The mean robot's energy consumption per step, considering all the steps.
- *Energy no failures*: The mean robot's energy consumption per step, considering only the successful steps.
- *Energy per output*: The mean robot's energy consumption per output. It is calculated as the total energy consumption considering all the steps divided by the number of successful steps.

The safety metrics evaluate how safe the robot's navigation is and are taken from [8]. For the sake of simplicity, since there are already many offloading metrics, only the most relevant metrics from [8] are used:

- *Critical zone*: The percentage of time with the closest obstacle in the critical (red) zone.
- *Warning zone*: The percentage of time with the closest obstacle in the warning (yellow) zone.
- *Safe zone*: The percentage of time with no obstacle in the warning and critical zones (i.e., the closest obstacle is in the green zone or outside it).

- *Risk* $\times$ *speed*: The risk value is multiplied by the robot's speed and stored. Then, the mean value is calculated to get this metric.

It is worth mentioning that the RL safety-oriented task-offloading agent affects the offloading metrics directly and the safety metrics indirectly. Indeed, latency and accuracy of the scene understanding impact the quality of the risk management process, which determines the safety of the navigation.

4.3 Results

The results of the experiments for the scenarios with one and two robots are reported in tables 4.2 and 4.3, respectively. For the latter scenario, since the two robots performed very similarly, only the outcome for one of them is shown. For completeness, appendix A contains the results for the other one. In line with the evaluation framework described in section 4.2, the tables include offloading and safety metrics (separated by a horizontal line) for the baselines as well as the RL agents (DQN-2 and DQN-3). The rest of this section provides a systematic analysis.

4.3.1 Analysis of task offloading

The results showed that the edge-only policy, which always offloads independently of the network state, was not appropriate for the HRC scenarios. In the single-robot scenario, while showing the potential of edge computing with the best *latency no failures*, the high percentage of failures ($65\,\%$ of *success*) made it the worst agent in terms of *latency* and *latency per output*. After adding another robot always offloading, this baseline completely failed at sharing the network resources ($13.41\,\%$ of *success* and $14.21\,\mathrm{s}$ of *latency per output*). These findings correct the previous results in [8], where this naive policy was tested in a network without other robots or devices creating congestion, and confirm the motivation for this degree project.

On the opposite side, the robot-only policy achieved a relatively high latency ($1.06\,\mathrm{s}$), which may be inappropriate in dangerous situations. It also had the highest energy consumption, implying that the robot must be charged more often than the other agents. Besides this, the main limitation of this naive agent is that it always uses the less accurate onboard model ($0\,\%$ of *edge output*) without exploiting the edge. Thus, the robot-only policy is not the best approach.

Table 4.2: Results for the scenario with a single robot, grouped by type of metric (offloading and safety). The best results for each metric are highlighted in bold.

Metric	Edge-only	Robot-only	Random	w/o RM	DQN-2	DQN-3
Latency (s)	1.18 ± 0.06	1.06 ± 0.00	1.06 ± 0.04	-	0.91 ± 0.02	$\mathbf{0.55 \pm 0.04}$
Latency no failures (s)	0.74 ± 0.01	1.06 ± 0.00	0.90 ± 0.02	-	0.81 ± 0.02	$\mathbf{0.48 \pm 0.03}$
Latency per output (s)	1.83 ± 0.21	1.06 ± 0.00	1.25 ± 0.08	-	0.99 ± 0.02	$\mathbf{0.58 \pm 0.04}$
Risk $\times$ latency (s)	1.72 ± 1.69	1.60 ± 1.08	1.42 ± 1.36	-	1.40 ± 1.25	$\mathbf{0.90 \pm 1.20}$
Success (%)	64.95 ± 4.19	$\mathbf{100.00 \pm 0.00}$	85.46 ± 2.44	-	91.85 ± 0.28	95.25 ± 0.64
Edge output (%)	$\mathbf{64.95 \pm 4.19}$	0.00 ± 0.00	35.58 ± 2.82	-	54.62 ± 3.44	47.15 ± 8.34
Valid scene graph (%)	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	-	$\mathbf{100.00 \pm 0.00}$	86.09 ± 1.93
Energy (J)	$\mathbf{0.11 \pm 0.01}$	10.52 ± 0.03	5.32 ± 0.14	-	3.99 ± 0.36	2.60 ± 0.25
Energy no failures (J)	$\mathbf{0.06 \pm 0.01}$	10.52 ± 0.03	6.17 ± 0.25	-	4.31 ± 0.39	2.71 ± 0.27
Energy per output (J)	$\mathbf{0.17 \pm 0.04}$	10.52 ± 0.03	6.23 ± 0.26	-	4.35 ± 0.39	2.73 ± 0.27
Critical zone (%)	2.16 ± 1.76	2.13 ± 3.19	3.62 ± 4.12	2.06 ± 1.41	1.45 ± 2.03	$\mathbf{0.83 \pm 2.46}$
Warning zone (%)	14.18 ± 0.84	$\mathbf{13.89 \pm 1.02}$	14.30 ± 0.76	17.48 ± 1.42	14.21 ± 1.32	14.65 ± 1.77
Safe zone (%)	83.67 ± 2.18	83.98 ± 3.74	82.08 ± 3.53	81.46 ± 2.24	84.34 ± 2.42	$\mathbf{84.52 \pm 1.75}$
Risk $\times$ speed (m/s)	0.69 ± 0.03	0.70 ± 0.03	0.71 ± 0.08	0.72 ± 0.03	0.71 ± 0.04	$\mathbf{0.68 \pm 0.09}$

Table 4.3: Results for the first robot in the scenario with two robots, grouped by type of metric (offloading and safety). The best results for each metric are highlighted in bold.

Metric	Edge-only	Robot-only	Random	w/o RM	DQN-2	DQN-3
Latency (s)	1.89 ± 0.01	1.06 ± 0.00	1.38 ± 0.03	-	0.95 ± 0.02	$\mathbf{0.67 \pm 0.03}$
Latency no failures (s)	1.12 ± 0.05	1.06 ± 0.00	1.04 ± 0.01	-	0.84 ± 0.02	$\mathbf{0.54 \pm 0.02}$
Latency per output (s)	14.21 ± 1.34	1.06 ± 0.00	2.15 ± 0.15	-	1.02 ± 0.04	$\mathbf{0.73 \pm 0.04}$
Risk $\times$ latency (s)	2.91 ± 1.98	1.60 ± 1.10	2.18 ± 1.69	-	1.45 ± 1.34	$\mathbf{1.17 \pm 1.50}$
Success (%)	13.41 ± 1.18	$\mathbf{100.00 \pm 0.00}$	64.55 ± 2.94	-	87.10 ± 1.11	91.59 ± 0.92
Edge output (%)	13.41 ± 1.18	0.00 ± 0.00	15.35 ± 3.00	-	22.05 ± 3.34	$\mathbf{23.18 \pm 3.58}$
Valid scene graph (%)	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	-	$\mathbf{100.00 \pm 0.00}$	79.94 ± 3.43
Energy (J)	$\mathbf{0.19 \pm 0.01}$	10.49 ± 0.03	5.34 ± 0.10	-	6.41 ± 0.39	3.90 ± 0.16
Energy no failures (J)	$\mathbf{0.09 \pm 0.01}$	10.49 ± 0.03	8.07 ± 0.39	-	7.71 ± 0.43	4.23 ± 0.20
Energy per output (J)	$\mathbf{1.40 \pm 0.27}$	10.49 ± 0.03	8.29 ± 0.41	-	7.80 ± 0.43	4.26 ± 0.20
Critical zone (%)	3.04 ± 1.15	2.45 ± 2.38	3.03 ± 1.07	1.44 ± 1.23	1.61 ± 3.07	$\mathbf{1.13 \pm 0.98}$
Warning zone (%)	18.97 ± 1.64	$\mathbf{14.14 \pm 1.38}$	14.78 ± 1.37	16.97 ± 1.27	14.97 ± 2.21	14.33 ± 1.09
Safe zone (%)	77.99 ± 2.61	83.41 ± 2.62	82.19 ± 1.53	81.59 ± 2.52	83.42 ± 3.20	$\mathbf{84.54 \pm 2.04}$
Risk $\times$ speed (m/s)	0.68 ± 0.06	0.67 ± 0.02	0.70 ± 0.04	0.73 ± 0.04	$\mathbf{0.64 \pm 0.03}$	0.66 ± 0.03

DQN-2 dynamically decides whether to offload the scene understanding or to run it locally by considering, among other things, the network state. In the single-robot scenario, the results show that it avoided to use the edge when the network was congested, limiting the number of failures ($+27\,\%$ of *success* compared to the edge-only policy). Consequently, this RL agent outperformed all the baselines in terms of *latency* and *latency per output* ($-45\,\%$ of *latency per output* compared to the edge-only policy). It is worth highlighting that performing better than the random baseline indicates that DQN-2 learned something useful from its experience and confirms the soundness of the designed reward function.

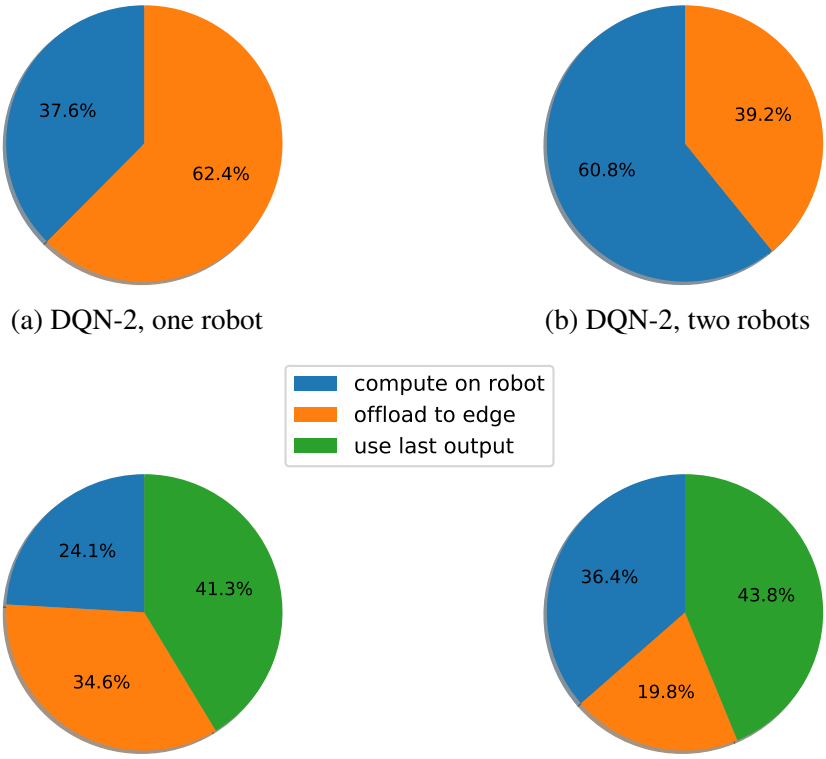
In the scenario with two robots, DQN-2 continued to work correctly, in contrast with the edge-only naive agent, and still outperformed the baselines. For better understanding this, it is useful to observe figure 3.2, which shows the distribution of actions taken by the RL agents. When another robot was added to the warehouse, both DQN-2 and DQN-3 reduced the edge usage, so they adapted to the network congestion and fairly shared the resources.

Since the main goal of this degree project is to have safety-oriented decision-making, it is relevant to analyze the agents' behavior by keeping an eye on the risk value. Figure 4.4 illustrates the joint distribution of risk value and latency for baselines and DQN-2[17]. In contrast with the baselines, which did not show any risk-dependent logic, DQN-2 adapted the latency to the safety requirements, offloading the task to the MEC server only when necessary and avoiding unnecessary congestion in the network. In particular, when the risk was low, the agent allowed local computation, which resulted in a relatively high delay (around $1\,\mathrm{s}$) but did not congest the network. Instead, in more hazardous situations, it tried to reduce the latency, often successfully. This evaluation is quantified by the *risk* $\times$ *latency* metric, which was reduced by DQN-2 ($-48\,\%$ compared to the edge-only policy in the single-robot scenario).

The same conclusion can be drawn by looking at figure 4.5, which shows the joint distribution of risk value and edge output (Boolean value from substate S3). Indeed, when the risk value is mid-high the agent preferred the more accurate model on the edge ($edge\ output = 1$), whereas accepting local outputs ($edge\ output = 0$) in safe situations.

DQN-3 differs from DQN-2 only for the possibility of skipping the computation and reusing the last output (action A3). While this action is beneficial in terms of latency and success rate, as it provides the scene graph

[17] DQN-3 is omitted because the plot would be confused by action A3, which gives zero latency.



(a) DQN-2, one robot

(b) DQN-2, two robots

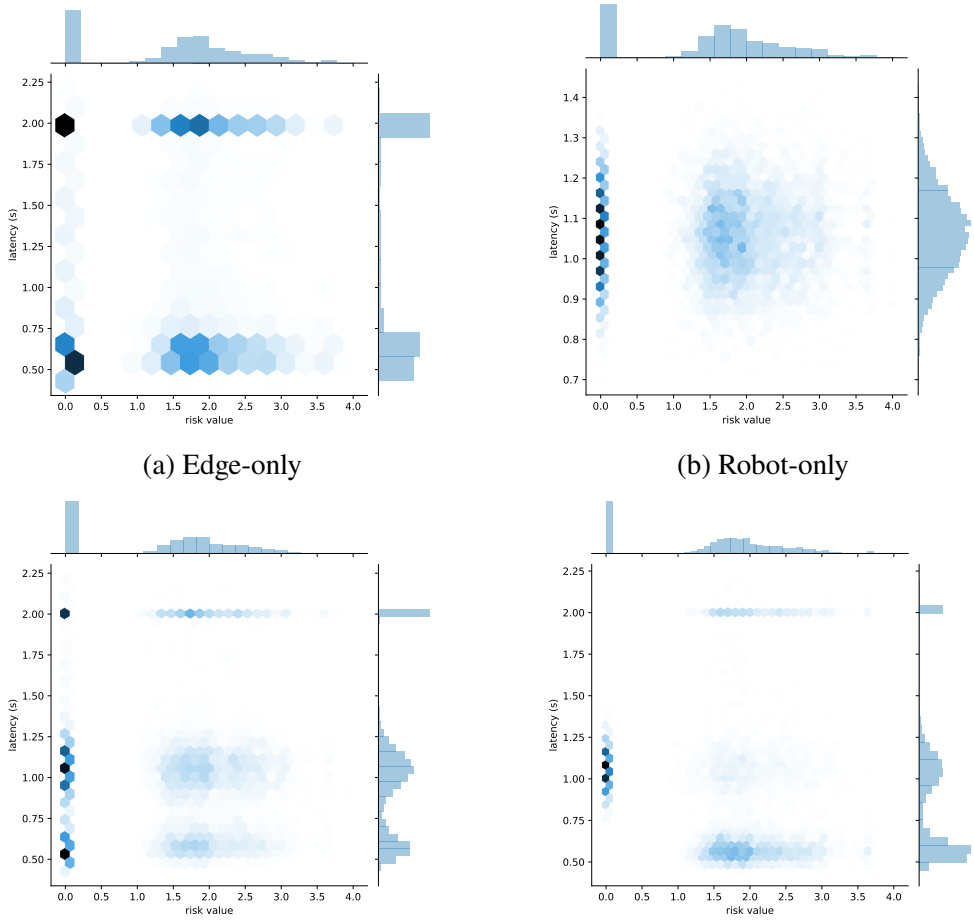
(c) DQN-3, one robot

(d) DQN-3, two robots

Figure 4.3: Distribution of actions for DQN-2 and DQN-3 in scenarios with one and two robots.

immediately, it should be used wisely. The results indicate that DQN-3 learned to exploit such an action, reusing the last scene graph only when appropriate ($86.09\,\%$ and $79.94\,\%$ of valid scene graph in the scenarios with one and two robots, respectively). Interestingly, as can be seen in figure 4.3, action A3 turned out to be the dominant choice (more than $40\,\%$ of the times). DQN-3 performed even better than DQN-2 in terms of latency, especially in the scenario with two robots, where skipping the computation also helps to avoid network congestion ($+4\,\%$ of *success* and $-40\,\%$ of *latency per output*).

Another interesting analysis is about understanding how DQN-3 decides to skip the computation based on the environment state. In this regard, figure 4.6 represents the scatter plot of risk value and temporal coherence, while differentiating between computation performed (actions A1 and A2) and skipped (action A3). As can be seen, DQN-3 learned a safety-based decision boundary for skipping the computation, with a threshold for the temporal coherence that slightly increases as the risk value grows. It is worth remarking



(a) Edge-only

(b) Robot-only

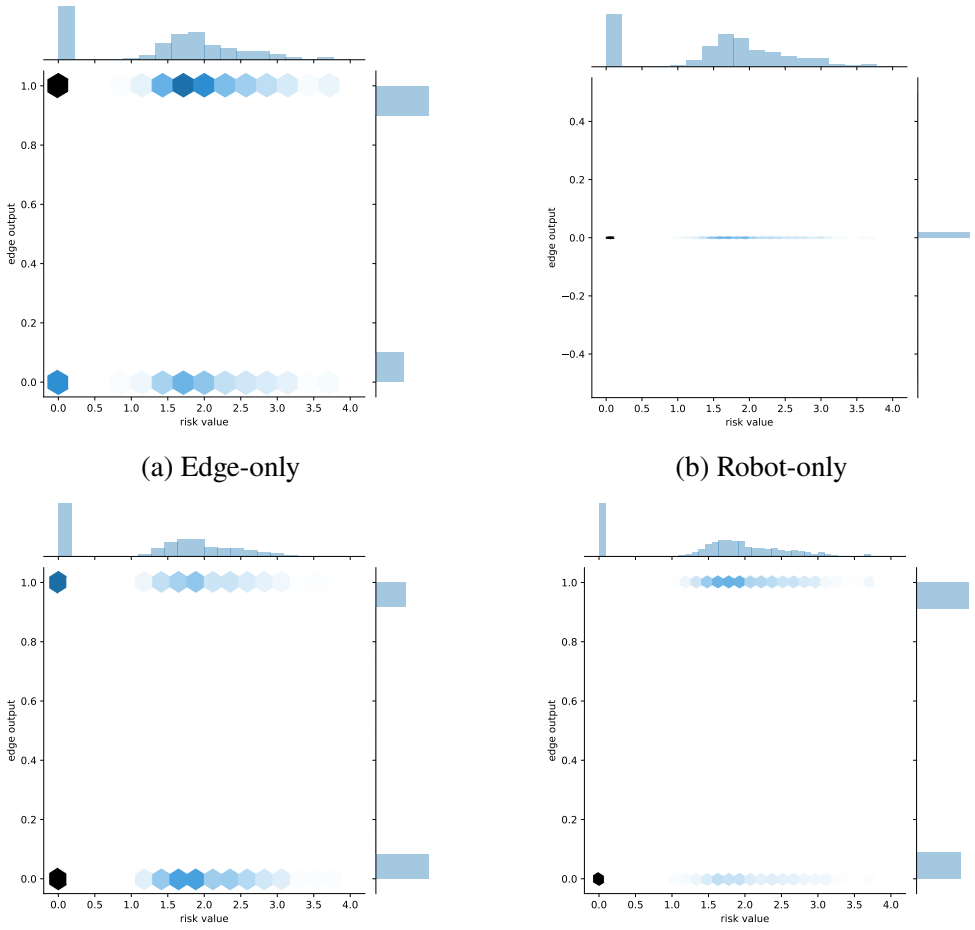
(c) Random

(d) DQN-2

Figure 4.4: Joint distribution of risk value and latency for DQN-2 and baselines in the scenario with one robot. The intensity of the blue color indicates the frequency of the joint value (i.e., the darker the blue, the more frequent), while the histograms on the axes represent the marginal distributions.

that the robot found this out on its own, as the reward function does not use the temporal coherence directly but only provides a means to understand the quality of the scene graph. This represents an example of one of the advantages of RL: learning from goal-oriented rewards.

Lastly, a positive side effect of edge computing is on energy consumption, as sending data on the network usually consumes less than using the CPU for local computation. Indeed, in terms of energy, DQN-2 resulted better than the robot-only ($-59\,\%$ and $-26\,\%$ of *energy per output* in the scenario with



(a) Edge-only

(b) Robot-only

(c) Random

(d) DQN-2

Figure 4.5: Joint distribution of risk value and edge output (Boolean value from substate S3) for DQN-2 and baselines in the scenario with one robot. The intensity of the blue color indicates the frequency of the joint value (i.e., the darker the blue, the more frequent), while the histograms on the axes represent the marginal distributions.

one and two robots, respectively) and random ($-30\,\%$ and $-6\,\%$ of *energy per output* in the scenario with one and two robots, respectively) baselines. Skipping the computation played an important role also in this case, with DQN-3 further reducing the energy consumption compared to DQN-2 ($-37\,\%$ and $-45\,\%$ of *energy per output* in the scenario with one and two robots, respectively).

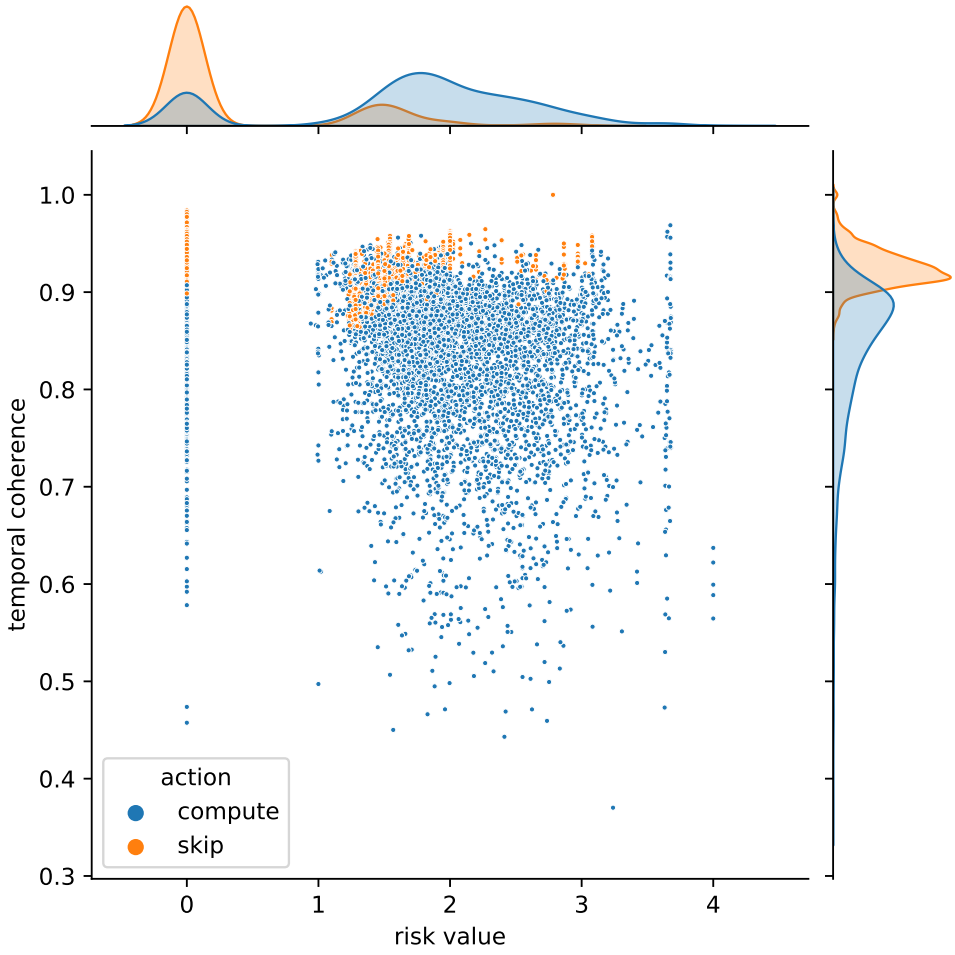


Figure 4.6: Scatter plot (i.e., there is one point for each action taken) of risk value and temporal coherence for DQN-3 in the scenario with one robot. The plot differentiates computation performed (actions A1 and A2) and skipped (action A3) using different colors. The marginal distributions are also shown on the axes as Gaussian.

4.3.2 Analysis of safety

The ultimate goal of this degree project is to intelligently offload the scene understanding to enhance safety in the warehouse. Indeed, the scene understanding offloading impacts safety both for latency and accuracy of the scene graph. However, the reader is reminded that the implementation in the simulated environment described in section 3.2 produces a perfect scene graph both for edge and local computation. So, in the experiments, safety is affected

only by latency. A real environment in which the edge provides more accurate scene graphs is envisioned to result in larger differences between robot-only policy and the RL agents, but such experiments are left as future work.

In the single-robot scenario, the baseline without risk mitigation performed the worst for all the safety metrics, meaning that the safety framework had a positive effect also with the naive offloading agents. For the scenario with two robots, instead, an interesting result can be seen: the edge-only policy compromised safety compared to not having risk mitigation ($-3.6\,\%$ of *time in safe zone*). This finding is reasonable because the risk mitigation can wrongly adjust the navigation when based on delayed information. The robot-only policy, instead, which provides the scene graph at a constant rate, turned out to be the best among the naive agents in both scenarios (about $+2\,\%$ of *time in safe zone*).

However, the RL agents outperformed all the baselines in terms of all the safety metrics in both scenarios (up to $+7\,\%$ of *time in safe zone* and up to $-10\,\%$ of *risk* $times$ *speed*), once again validating the soundness of the solution. More specifically, DQN-3 performed the best. The advantage of adding action A3 over DQN-2 was subtle in the single-robot scenario ($+0.2\,\%$ of *time in safe zone*), but became more relevant with two robots ($+1.1\,\%$ of *time in safe zone*).

Chapter 5

Conclusions and future work

This chapter concludes the thesis. Section 5.1 briefly reviews the solution and provides general conclusions, with links to the objectives presented in chapter 1. Section 5.2 describes the limitations of the results, while section 5.3 suggests future work. Finally, section 5.4 reflects on the impact of the work and discusses ethical and societal aspects.

5.1 Discussion

This degree project aims at intelligently using MEC in a HRC scenario where constrained-hardware mobile robots need the real-time execution of a safety framework that includes a computationally expensive task, namely the scene understanding. For guaranteeing timely outputs of such a task, three possible actions were identified: edge computation of the complex task, local computation of a less accurate version of the task, and computation skip with reuse of the previous output. DQN was then employed to learn choosing among these actions rationally, based on network and safety aspects. For achieving this, the problem was formalized as an RL environment, utilizing meaningful KPIs to monitor the network and safety states (throughput, RTT, and risk value) and formulating a specific reward function. The proposed solution was designed and implemented as a modular system and integrated with the previously developed safety framework. Specific simulated scenarios were also realized to test and evaluate the solution using a set of metrics and baselines. The extensive experiments validated the soundness of the method and allowed drawing some general conclusions, discussed as follows.

The solution provides dynamic decision-making to use the edge only when suitable. In particular, the agent offloads the computation to the MEC

server only when the network performance is not degraded and when an unsafe situation is encountered. When such conditions are not satisfied, the simplified version of the task providing lower accuracy runs locally on the robot. Therefore, the solution adapts latency and accuracy to the safety requirements, meaning that the robot avoids using the edge when the risk is low, letting other nodes use it and thus wisely sharing the network resources. Furthermore, given that the primary inputs to the scene understanding are images from the robot's camera, the inherently present temporal coherence is exploited to skip the computation and reuse the previous output when suitable, providing benefits in terms of latency, energy consumption, and network congestion.

The ultimate goal of this project, which is the safety enhancement in the HRC scenario employing MEC, was successfully achieved. Indeed, the learned policy executes the scene understanding faster than the previously used edge-only policy and guarantees low latency and high accuracy in risky situations. Consequently, the whole safety framework runs at a higher frequency, especially when the situation is hazardous.

Another key feature of the solution is that it is learning-based. The robot learns its policy by interacting with the (simulated) environment using RL, and the reward function allowing this is one of the significant contributions of this work. Therefore, there are no hard-coded rules which usually rely on domain-expert knowledge and are not robust to changes in the environment. Moreover, even though this project considered the specific case of the scene understanding, the solution may be generalized and applied to other safety-related tasks.

5.2 Limitations

As discussed throughout the thesis, the simulated environment imposed some constraints and forced some choices for the experiments. These constraints also affect and limit the results, as described below.

The limitation about the number of simulated robots, which was bounded to 2, was partially compensated by employing virtual congesting nodes creating more traffic over the network. However, the robots have a different network usage pattern because they send data for shorter periods. In this regard, the solution was validated with two robots managing to fairly share the available resources in a network congested by other devices. However, it should be further tested in a scenario with more robots. These experiments could also help determine the upper bound of the number of robots in the

warehouse that can share the network while achieving safe operations. Such a value would also depend on network technology and warehouse scenario (e.g., number of human workers).

The other limitation concerns the implementation of the scene understanding. In order to achieve the correct latency in the simulated environment, the scene graph was generated by querying V-REP. However, this always provides a perfect output and does not simulate the different accuracy of local and edge computing, so the results do not consider such a distinction. Significantly, this underestimates the safety improvement of edge computing compared to local computing. Therefore, it is envisioned that the solution exploiting MEC provides more benefits in terms of safety compared to local computing.

5.3 Future work

The most straightforward future work is to test the solution with other RL algorithms. For example, CEM and SARSA could be tested with minimal effort because they are provided by Keras-RL and support discrete action space and continuous state space, like DQN.

Concerning the network, although in this work Wi-Fi 802.11g was chosen to run ns-3 effectively in real-time, it is possible to test other technologies with simulations on more powerful PCs or in real environments. The most interesting one is 5G, which is envisioned as the most used in industry 4.0. Theoretically, 5G can support a higher number of robots in the warehouse and can achieve lower latency than Wi-Fi 802.11g, leading to enhanced safety.

Another relevant future work is to test the solution in a real environment. This extension would require an expensive testbed including many robots and dedicated network infrastructure (e.g., a 5G cell and a small data center) as well as training YOLACT for the onboard scene understanding. Moreover, the $ExtractSceneGraph$ function in algorithm 3.1 would need to be redefined because the extraction from V-REP would not be possible. For example, it might build a simplified scene graph with just distance and direction by using sensor data from a LiDAR.

The suggestions mentioned above are essentially experiments to further validate the solution. It is also possible to extend the solution by adding explainability. Indeed, explainable RL [75] is an emerging research area and, for example, it was applied to the risk mitigation module of the safety framework presented in section 2.1, achieving relevant results [76].

Finally, a different approach could be realized by using a centralized agent which collects data from all the robots in the warehouse and decides which

ones should offload. This solution would be similar to the related work based on game theory, thus more complex to implement for the reasons described in section 2.5, but could still be realized and compared to the results of this work.

5.4 Ethical and societal aspects

This work contributes to the realization of a safe HRC in industry 4.0 by providing an AI-based solution for MEC usage. Since this degree project is related to AI and safety, which are the subject of numerous debates, it is relevant to discuss its impact from an ethical and societal point of view.

Without a doubt, there are ethical issues that can occur in the HRC scenario considered in this work. For example, what if two robots are both close to humans? Should the robot surrounded by more people have precedence to offload? Or should this decision be based on the age of the people? Furthermore, who would be responsible if a robot harms a human because it decided not to offload? Unfortunately, questions like these are hard to answer, and there are still no laws and regulations nowadays.

On the other hand, HRC has a significant impact on society. The whole population can benefit from the improved quality of industrial production. Also, the industrial employees can avoid doing repetitive (that is a synonym of boring for most people) tasks, so the quality of work is highly improved. By properly guaranteeing safety, this shift in the work-life can happen with human workers trusting their new artificial workmates.

Lastly, regarding sustainability, the solution proposed in this project considers skipping the computation and reusing the previous output and chooses this action whenever possible. This feature saves energy and is therefore environmentally friendly.

References

[1] N. Shan, Y. Li, and X. Cui, "A Multilevel Optimization Framework for Computation Offloading in Mobile Edge Computing," *Mathematical Problems in Engineering*, vol. 2020, pp. 1–17, Jun. 2020. DOI: 10.1155/2020/4124791

[2] "Exchange Data with ROS Publishers and Subscribers - MATLAB & Simulink," https://www.mathworks.com/help//ros/ug/exchange-data-with-ros-publishers-and-subscribers.html.

[3] "Call and Provide ROS Services - MATLAB & Simulink," https://www.mathworks.com/help/ros/ug/call-and-provide-ros-services.html.

[4] S. Robla-Gómez, V. M. Becerra, J. R. Llata, E. González-Sarabia, C. Torre-Ferrero, and J. Pérez-Oria, "Working Together: A Review on Safe Human-Robot Collaboration in Industrial Environments," *IEEE Access*, vol. 5, pp. 26 754–26 773, 2017. DOI: 10.1109/ACCESS.2017.2773127

[5] R. Inam, E. Fersman, K. Raizer, R. Souza, A. Nascimento, and A. Hata, "Safety for automated warehouse exhibiting collaborative robots," in *Safety and Reliability – Safe Societies in a Changing World*, Jun. 2018, pp. 2021–2028. ISBN 978-1-351-17466-4

[6] R. Inam, K. Raizer, A. Hata, R. Souza, E. Forsman, E. Cao, and S. Wang, "Risk Assessment for Human-Robot Collaboration in an automated warehouse scenario," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, Sep. 2018. DOI: 10.1109/ETFA.2018.8502466. ISSN 1946-0759 pp. 743–751.

[7] A. Hata, R. Inam, K. Raizer, S. Wang, and E. Cao, "AI-based Safety Analysis for Collaborative Mobile Robots," in *2019 24th*

IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Sep. 2019. DOI: 10.1109/ETFA.2019.8869263. ISSN 1946-0759 pp. 1722–1729.

[8] A. Terra, H. Riaz, K. Raizer, A. Hata, and R. Inam, "Safety vs. Efficiency: AI-Based Risk Mitigation in Collaborative Robotics," in *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, Apr. 2020. DOI: 10.1109/ICCAR49639.2020.9108037. ISSN 2251-2446 pp. 151–160.

[9] H. Riaz, A. Terra, K. Raizer, R. Inam, and A. Hata, "Scene Understanding for Safety Analysis in Human-Robot Collaborative Operations," in *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, Apr. 2020. DOI: 10.1109/ICCAR49639.2020.9108083. ISSN 2251-2446 pp. 722–731.

[10] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, Sep. 2019. DOI: 10.1016/j.sysarc.2019.02.009

[11] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A Survey on Mobile Edge Computing: The Communication Perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, Fourthquarter 2017. DOI: 10.1109/COMST.2017.2745201

[12] E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov. 2013. DOI: 10.1109/IROS.2013.6696520. ISSN 2153-0866 pp. 1321–1326.

[13] "Robot simulator CoppeliaSim: Create, compose, simulate, any robot - Coppelia Robotics," https://www.coppeliarobotics.com/.

[14] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds. Berlin, Heidelberg: Springer, 2010, pp. 15–34. ISBN 978-3-642-12331-3

[15] nsnam, "Ns-3," https://www.nsnam.org/.

[16] L. Žlajpah, "Simulation in robotics," *Mathematics and Computers in Simulation*, vol. 79, no. 4, pp. 879–897, Dec. 2008. DOI: 10.1016/j.matcom.2008.02.017

[17] R. Inam, N. Schrammar, K. Wang, A. Karapantelakis, L. Mokrushin, A. V. Feljan, and E. Fersman, "Feasibility assessment to realise vehicle teleoperation using cellular networks," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, Nov. 2016. DOI: 10.1109/ITSC.2016.7795920. ISSN 2153-0017 pp. 2254–2260.

[18] Bonsai, "Simulators: The Key Training Environment for Applied Deep Reinforcement Learning," Feb. 2018.

[19] D. Vassis, G. Kormentzas, A. Rouskas, and I. Maglogiannis, "The IEEE 802.11g standard for high data rate WLANs," *IEEE Network*, vol. 19, no. 3, pp. 21–26, May 2005. DOI: 10.1109/MNET.2005.1453395

[20] Y. Xiao, "IEEE 802.11n: Enhancements for higher throughput in wireless LANs," *IEEE Wireless Communications*, vol. 12, no. 6, pp. 82–91, Dec. 2005. DOI: 10.1109/MWC.2005.1561948

[21] "TurtleBot," https://www.turtlebot.com/.

[22] "Interbotix Turtlebot 2i Mobile ROS Platform," https://www.trossenrobotics.com/interbotix-turtlebot-2i-mobile-ros-platform.aspx.

[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, vol. 3, Jan. 2009.

[24] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2961–2969.

[25] J. S. Ward and A. Barker, "A Cloud Computing Survey: Developments and Future Trends in Infrastructure as a Service Computing," *arXiv:1306.1394 [cs]*, Jun. 2013.

[26] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A Survey of Research on Cloud Robotics and Automation," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 398–409, Apr. 2015. DOI: 10.1109/TASE.2014.2376492

[27] J. Wan, S. Tang, H. Yan, D. Li, S. Wang, and A. V. Vasilakos, "Cloud robotics: Current status and open issues," *IEEE Access*, vol. 4, pp. 2797–2807, 2016. DOI: 10.1109/ACCESS.2016.2574979

[28] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., ser. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018. ISBN 978-0-262-03924-6

[29] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, Sep. 2013. DOI: 10.1177/0278364913495721

[30] C. Yu, J. Liu, and S. Nemati, "Reinforcement Learning in Healthcare: A Survey," *arXiv:1908.08796 [cs]*, Apr. 2020.

[31] I. Szita, "Reinforcement Learning in Games," in *Reinforcement Learning: State-of-the-Art*, ser. Adaptation, Learning, and Optimization, M. Wiering and M. van Otterlo, Eds. Berlin, Heidelberg: Springer, 2012, pp. 539–577. ISBN 978-3-642-27645-3

[32] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[33] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An Introduction to Deep Reinforcement Learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018. DOI: 10.1561/2200000071

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236

[35] "ROS.org | Powering the world's robots."

[36] "Documentation - ROS Wiki," http://wiki.ros.org/.

[37] "Standard C++," https://isocpp.org/.

[38] "Welcome to Python.org," https://www.python.org/.

[39] "Navigation - ROS Wiki," http://wiki.ros.org/navigation.

[40] "The Programming Language Lua," https://www.lua.org/.

[41] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, Sep. 2004. DOI: 10.1109/IROS.2004.1389727 pp. 2149–2154 vol.3.

[42] "Gazebo," http://gazebosim.org/.

[43] "Ns-3 Tutorial — Tutorial," https://www.nsnam.org/docs/tutorial/html.

[44] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540 [cs]*, Jun. 2016.

[45] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms," https://gym.openai.com.

[46] M. Plappert, "Keras-rl," 2016.

[47] "Keras: The Python deep learning API," https://keras.io/.

[48] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107496, Dec. 2020. DOI: 10.1016/j.comnet.2020.107496

[49] B. Cao, L. Zhang, Y. Li, D. Feng, and W. Cao, "Intelligent Offloading in Multi-Access Edge Computing: A State-of-the-Art Review and Framework," *IEEE Communications Magazine*, vol. 57, no. 3, pp. 56–62, Mar. 2019. DOI: 10.1109/MCOM.2019.1800608

[50] H. Guo and J. Liu, "Collaborative Computation Offloading for Multiaccess Edge Computing Over Fiber–Wireless Networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4514–4526, May 2018. DOI: 10.1109/TVT.2018.2790421

[51] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digital Communications and Networks*, vol. 5, no. 1, pp. 10–17, Feb. 2019. DOI: 10.1016/j.dcan.2018.10.003

[52] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016. DOI: 10.1109/TNET.2015.2487344

[53] L. Yang, H. Zhang, X. Li, H. Ji, and V. C. M. Leung, "A Distributed Computation Offloading Strategy in Small-Cell Networks Integrated With Mobile Edge Computing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2762–2773, Dec. 2018. DOI: 10.1109/TNET.2018.2876941

[54] H. Guo, J. Liu, J. Zhang, W. Sun, and N. Kato, "Mobile-Edge Computation Offloading for Ultradense IoT Networks," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4977–4988, Dec. 2018. DOI: 10.1109/JIOT.2018.2838584

[55] L. Li, K. Ota, and M. Dong, "Sustainable CNN for Robotic: An Offloading Game in the 3D Vision Computation," *IEEE Transactions on Sustainable Computing*, vol. 4, no. 1, pp. 67–76, Jan. 2019. DOI: 10.1109/TSUSC.2018.2844348

[56] S. Chinchali, A. Sharma, J. Harrison, A. Elhafsi, D. Kang, E. Pergament, E. Cidon, S. Katti, and M. Pavone, "Network Offloading Policies for Cloud Robotics: A Learning-based Approach," *arXiv:1902.05703 [cs]*, Feb. 2019.

[57] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep Reinforcement Learning for Online Computation Offloading in Wireless Powered Mobile-Edge Computing Networks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2581–2593, Nov. 2020. DOI: 10.1109/TMC.2019.2928811

[58] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, Apr. 2004. DOI: 10.1109/TIP.2003.819861

[59] Bonsai, "Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions," https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0, Nov. 2017.

[60] Z. Chen and X. Wang, "Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach," *EURASIP Journal on Wireless Communications and Networking*, vol. 2020, no. 1, p. 188, Sep. 2020. DOI: 10.1186/s13638-020-01801-6

[61] M. Tang and V. W. Wong, "Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems," *IEEE Transactions on Mobile Computing*, pp. 1–1, 2020. DOI: 10.1109/TMC.2020.3036871

[62] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized Computation Offloading Performance in Virtual Edge Computing Systems Via Deep Reinforcement Learning," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019. DOI: 10.1109/JIOT.2018.2876279

[63] I. Szita and A. Lörincz, "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, Dec. 2006. DOI: 10.1162/neco.2006.18.12.2936

[64] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "YOLACT: Real-Time Instance Segmentation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 9157–9166.

[65] "Aim-uofa/AdelaiDet," Adelaide Intelligent Machines (AIM) Group, Jul. 2021.

[66] D. Bolya, "Dbolya/yolact," Jul. 2021.

[67] "OpenVINO™ Toolkit Overview - OpenVINO™ Toolkit," https://docs.openvinotoolkit.org/latest/index.html.

[68] "UserBenchmark: Intel Celeron J3455 vs Core i7-8650U," https://cpu.userbenchmark.com/Compare/Intel-Core-i7-8650U-vs-Intel-Celeron-J3455/m353957vsm200485.

[69] "Intel Celeron J3455 processor review: CPU specs, performance benchmarks," https://askgeek.io/en/cpus/Intel/Celeron-J3455.

[70] "8 reasons to turn down the transmit power of your Wi-Fi - Metis.fi," https://metis.fi/en/2017/10/txpower/.

[71] A. F. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," *arXiv:1803.08375 [cs, stat]*, Feb. 2019.

[72] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017.

[73] "Ns-3: Examples/wireless/wifi-tcp.cc Source File," https://www.nsnam.org/doxygen/wifi-tcp_8cc_source.html.

[74] M. Lacage and T. R. Henderson, "Yet another network simulator," in *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*, ser. WNS2 '06. New York, NY, USA: Association for Computing Machinery, Oct. 2006. DOI: 10.1145/1190455.1190467. ISBN 978-1-59593-508-3 pp. 12–es.

[75] E. Puiutta and E. M. S. P. Veith, "Explainable Reinforcement Learning: A Survey," in *Machine Learning and Knowledge Extraction*, ser. Lecture Notes in Computer Science, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-57321-8_5. ISBN 978-3-030-57321-8 pp. 77–95.

[76] A. Iucci, *Explainable Reinforcement Learning for Risk Mitigation in Human-Robot Collaboration Scenarios*. DiVA, 2021.

Appendix A

Results for second robot

Table A.1 reports the results for the second robot in the scenario with two robots. As can be seen, they are very similar to table 4.3.

Table A.1: Results for the second robot in the scenario with two robots, grouped by type of metric (offloading and safety). The best results for each metric are highlighted in bold.

Metric	Edge-only	Robot-only	Random	w/o RM	DQN-2	DQN-3
Latency (s)	1.88 ± 0.02	1.06 ± 0.00	1.41 ± 0.05	-	0.93 ± 0.03	$\mathbf{0.60 \pm 0.09}$
Latency no failures (s)	1.13 ± 0.06	1.06 ± 0.00	1.06 ± 0.02	-	0.83 ± 0.03	$\mathbf{0.49 \pm 0.08}$
Latency per output (s)	13.23 ± 1.57	1.06 ± 0.00	2.17 ± 0.17	-	0.99 ± 0.06	$\mathbf{0.65 \pm 0.11}$
Risk $\times$ latency (s)	2.77 ± 2.04	1.56 ± 1.10	2.23 ± 1.70	-	1.52 ± 1.41	$\mathbf{1.05 \pm 1.45}$
Success (%)	14.41 ± 1.47	$\mathbf{100.00 \pm 0.00}$	63.98 ± 2.81	-	87.32 ± 1.73	92.59 ± 1.17
Edge output (%)	14.41 ± 1.47	0.00 ± 0.00	14.88 ± 2.99	-	22.20 ± 3.07	$\mathbf{22.83 \pm 8.11}$
Valid scene graph (%)	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	$\mathbf{100.00 \pm 0.00}$	-	$\mathbf{100.00 \pm 0.00}$	78.96 ± 5.71
Energy (J)	$\mathbf{0.19 \pm 0.01}$	10.49 ± 0.02	5.31 ± 0.12	-	6.72 ± 0.40	3.61 ± 0.49
Energy no failures (J)	$\mathbf{0.10 \pm 0.01}$	10.49 ± 0.02	8.11 ± 0.41	-	8.07 ± 0.41	3.87 ± 0.57
Energy per output (J)	$\mathbf{1.30 \pm 0.31}$	10.49 ± 0.02	8.33 ± 0.42	-	8.16 ± 0.41	3.90 ± 0.57
Critical zone (%)	4.37 ± 1.55	3.05 ± 2.98	3.58 ± 1.02	$\mathbf{1.54 \pm 1.19}$	2.34 ± 2.46	1.60 ± 3.28
Warning zone (%)	16.47 ± 1.54	13.90 ± 0.95	15.25 ± 1.39	16.47 ± 1.23	14.12 ± 0.61	$\mathbf{13.95 \pm 1.47}$
Safe zone (%)	79.16 ± 2.49	83.05 ± 3.77	81.17 ± 1.46	81.99 ± 2.55	83.53 ± 2.41	$\mathbf{84.45 \pm 2.72}$
Risk $\times$ speed (m/s)	0.65 ± 0.04	0.64 ± 0.00	0.72 ± 0.04	0.74 ± 0.04	0.64 ± 0.01	$\mathbf{0.60 \pm 0.07}$

TRITA -EECS-EX-2021:835