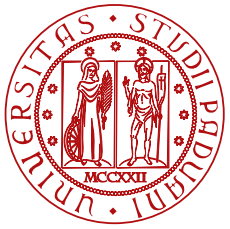


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Reinforcement learning in shared intelligence systems for mobile robots

Master Candidate

Miroljub Mihailovic

Student ID 2021445

Supervisor

Prof. Emanuele Menegatti

Co-supervisors

Dr. Gloria Beraldo

Prof. Loris Nanni

11 July 2022

Academic Year
2021/2022

To my parents
and friends

Abstract

In this thesis, we investigate how to integrate reinforcement learning in a shared intelligence system where the user's commands are equally fused with the robot's perception during the teleoperation of mobile robots. Specifically, we illustrate a new policy-based implementation suitable for navigating in unknown indoor environment with the ability to avoid collisions with dynamic and static obstacles. The aim of this thesis consists of extending the current system of shared intelligence based on numerous pre-defined policies with a new one based on Reinforcement Learning (RL).

To design this policy, an agent learns to reach a predefined goal by multiple trial and error interactions. To make the robot learn correct actions, a reward function is defined taking inspiration from the Attractive Potential Field (APF) and the state is computed through a pre-processing module that clusters the obstacles around the robot and finds the closest point with respect to the robot. Different clustering algorithms are analysed for establishing which of them is the most suitable for the purpose considering the real-time constraints required by the system.

Different model configurations are examined and trained in gazebo-based simulation scenarios and then evaluated in different navigation scenarios. In this way, we verified the reactive navigation behaviour of the agent with static and dynamic obstacles. The shared system combined with the new RL policy is tested and compared with the current state-of-the-art version, in a dedicated teleoperated experiment where an operator was required to interact with the robot by delivering high-level commands.

Sommario

In questa tesi, analizziamo come integrare il Reinforcement Learning in un sistema di intelligenza condivisa in cui i comandi dell'utente, con la percezione del robot, sono ugualmente considerati durante la teleoperazione di un robot mobile. Nello specifico, illustriamo una nuova implementazione policy-based adatta alla navigazione in ambienti interni sconosciuti in grado di evitare collisioni con ostacoli dinamici e statici. Lo scopo di questa tesi consiste nell'estendere l'attuale sistema di intelligenza condivisa basato su numerose policy predefinite con una nuova basata sul Reinforcement Learning (RL).
Per progettare questo tipo di policy, un agente impara a raggiungere un obiettivo predefinito mediante molteplici tentativi ed errori. Per far apprendere al robot le azioni corrette, viene definita una funzione di ricompensa ispirata all'Attractive Potential Field (APF) e lo stato viene calcolato attraverso un modulo di elaborazione che divide gli ostacoli attorno al robot e trova il punto più vicino rispetto a esso. Vengono analizzati diversi algoritmi di clustering per stabilire quale di essi sia più adatto allo scopo considerando i vincoli di tempo richiesti dal sistema.
Diverse configurazioni del modello vengono esaminate e addestrate in scenari di simulazione basati su Gazebo e successivamente valutate in diversi scenari di navigazione. In questo modo, è possibile verificare il comportamento reattivo dell'agente durante la navigazione in presenza di ostacoli statici e dinamici. Il sistema di intelligenza condivisa combinato con la nuova policy basata sul RL viene testato e confrontato con l'attuale approccio allo stato dell'arte, in un esperimento teleoperato in cui un operatore deve interagire con il robot fornendo comandi di alto livello.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Objective

Teleoperation refers to a remote operation for controlling a device. In several applications, a sort of intelligence is needed to overcome possible problems due to the lack of interpretability of commands (e.g., applications where the user's input is subject to noise). In particular, in this thesis we focus on teleoperating mobile robots through a new hybrid (model-based and learning-based) shared intelligence system. The current model-based approach [2] provides a system where the user and robot are synchronously combined for estimating trajectory without any information about the environment or final destination. This is done by modelizing the environment information for choosing the robot's destination through a fusion of multiple probability grids (*i.e.*, policy) equally based. Therefore, the purpose of this thesis is to increase the performance by adding one more policy for reaching the goal and at the same time avoiding possible collisions, and maintaining safe robot navigation. This is done by using a Reinforcement Learning (RL) model that differs from supervised learning in that it does not need labelled input/output pairings or the explicit correction of suboptimal behaviours. Consequentially, we have opted for a trial-and-error approach in a simulated environment where the model has been trained and tested with different configurations. Finally, for verifying the improvement of the new method, we have defined a experimental setup with 7 target positions in which the user controls the robot with the support of the new hybrid system. For evaluating which methods are the most suitable for the task, we have

considered some qualitative metrics, among which: the time for completing the path and the number of human interaction, the average distance from obstacles, the number of collisions and goals reached.

1.2 Thesis Structure

The thesis structure starts with an initial introduction (Chapter 2) of Reinforcement Learning description of some basic concepts that are necessary to understand which model is suitable for the objective defined previously. Furthermore, we list in Chapter 3 different Machine Learning libraries (Section 3.5, 3.4) exploited for training the model and clustering the obstacles. The other components (ROS, Gazebo and TIAGo) are used for defining the environment to train and test the model and the relative communications among the multiple components of the system. Chapter 4 shows in detail the current state-of-the-art shared intelligence implementation and explains how this thesis extends this system with the introduction of an RL component. In particular, we explain the main components among which: Cluster Analysis, Pre-Processing Module, Deep Deterministic Policy Gradient and Training Process. The first presents the examined cluster algorithms to define the current state for the RL based on the obstacles around the robot. The second introduces the processing performed to compute the distance between each cluster and the robot provided as input to the RL. The third defines the RL algorithm implemented with the reward function and their relative parameters chosen for maximizing the final training process. The last one illustrates the environments chosen for this thesis and the design selected to create the final model. In Chapter 5 we report the evaluation of the Cluster Analysis, Training Process and the relative tests in dynamic and static environment conditions. In addition to this, we evaluate the overall system, considering the policies defined in [2] combined with the new learning-based method. Chapter 6 concludes the thesis and illustrated the future works.

2

Reinforcement Learning

"Reinforcement Learning (RL) describes how to map situations to actions while maximizing the reward function" [3].
In other words, the agent must understand which actions are most appropriate by trying them and analysing their effectivity through the reward function. One of the most interesting and crucial characteristics is that the agent may not focus on forthcoming award, but also on future compensation.
The problem formulation is based on dynamical system theory as the optimal control of incompletely known Markov decision processes, which includes sensation, action and goal. We could see this procedure as an interaction cycle, Figure 2.1, where an agent is capable of capturing the state of its environment, and it ought to complete actions that affect the state until a predefined goal or sub-goal is achieved [3]. The actions influence the environment in two different ways. First, the internal changes that might be perceived in perspective interactions, in these situations the agent is in uncertainty, and it will not be able to accurately detect an internal change in the environment. Second, the environment reacts in an external form visible to the agent. Therefore, as a consequence, the environment's reaction could be a direct result of the last action made by the agent.
Evaluating the feedback is a complex task taking into consideration that is not trivial to interpret and categorize as good or bad decisions. For this reason, reinforcement learning relies on exhaustive sampling of the environment (*i.e.*, expressing a set of elements from the environment, which represent the data) through a "tabular" implementations. Such an implementation is feasible to ta-

ble a combination of several states and actions in such a way to expose possible situations in which the agent may encounter. Keep in mind that it is crucial not to see all probable situations, otherwise, the model may be poorly generalized [4].
One of RL's challenges that grow over time is the trade-off between exploration and exploitation.

- **Exploitation** implicates exploring a small area of the search space to achieve a better solution. This operation is equivalent to intensifying the search in the local neighbourhood.
- **Exploration**, on the other hand, involves a much larger portion of the research space, discovering other promising but unprocessed areas. This operation requires diversifying the search to avoid local minima.

RL agent prefers actions tried in the past to obtain more reward, however, to discover these kinds of actions it has to try that are not selected previously. The agent must understand several actions and progressively select those that appear better than others based on the reward function, each of them must be tried many times in such a way to gain a reliable estimate of the expected reward [3].

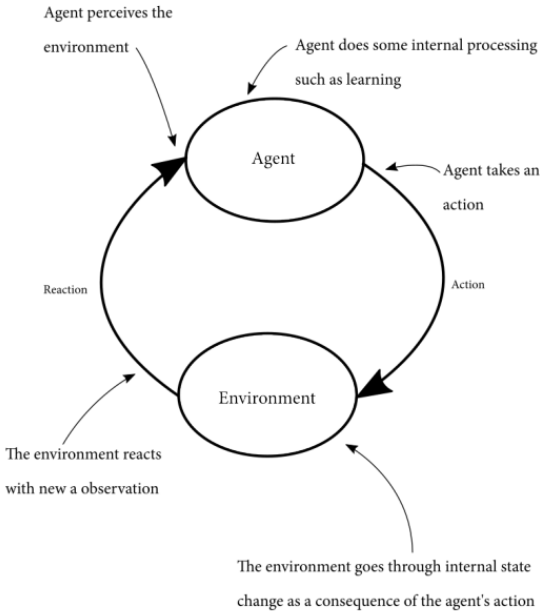


Figure 2.1: The reinforcement learning interaction cycle taken from [4].

2.1 Elements of Reinforcement Learning

There are four main elements that characterize Reinforcement Learning beyond the agent and environment discussed before. These are Policy, Reward Signal, Value Function and Model of the environment [3].

- Policy: defines the behaviour to be assumed by the agent at a given time in a stochastic way.
- Reward Signal: The reward signal defines which events are good and bad for the agent. This is done by the environment that sends a single number (*i.e.,* reward) to the agent at each stage. The agent maximizes the total reward that it receives over time.
- Value Function: It specifies the quality of the events (*i.e.,* good or bad) in future terms.
- Model of the environment: This allows us to make inferences about how the environment will behave. These kinds of techniques are classified as model-based and model-free, where the first one uses a planning model and the second one is based on trial and error learners. Further details will be described in the next section.

2.2 Reinforcement Learning Approaches

In this section, we will present the different categories of RL. Models are classified according to different categories and the model that has been chosen (for solving the RL problem) depends on the environment and the task that we have to accomplish.

2.2.1 Model-Based and Model-Free

Model-Based RL uses model and planning for solving RL problems. This means, that an agent attempts to comprehend its surrounding environment and

creates a model based on its interactions with them. The agent (or greedy agent) will try to perform an action that maximizes the reward, regardless of the consequences. More formally, it aims to calculate the probabilities of transitions (*i.e.*, given the state s, the probability to take an action a that moves the agent from the state s to s') $P^a_{s,s'} = p(s'|s,a)$ with the reward $R^a_{s,s'} = E\{r|s,s',a\}$ from state s to s' and relative to action a.
The **Model-Free** algorithm does not exploit the transition probability distribution (and the reward function) connected to the Markov decision process (MDP) which in RL represents the problem to be solved. In this situation, the model learns through a trial-and-error sequence [3].

2.2.2 On and Off Policy

Now we are going to introduce two RL classifications that differ in the model learning method.
Algorithms that analyze and enhance the same policy that is being used to choose actions for the agent are known as "**On-Policy** Learning Algorithms." This implies that for the choice of the next action, the agent will continue to use the same policy, which we will try to estimate and improve. In other words, it aims to reduce the separation between the target (*i.e.*, policy learnt by the agent) and behaviour (*i.e.*, policy for selecting the action) policies.
Algorithms for **Off-Policy** learning estimate and improve a policy that is not the same as the policy that is used for selecting an action. Therefore, Off-Policies have some benefits, among which continuing exploration while learning the best policy. On-Policy, in contrast, is taught as suboptimal policy [3].

2.2.3 Value-Based and Policy-Based Reinforcement Learning

We will explain two main approaches for solving an RL problem.
Value-Based RL learns the state or action-state value and acts by choosing the best action in the current state. There are many types of value functions, among which: **action-value** functions, which map each combination of state and action to a Q-value[1] and **state-value** functions that exclusively care about the current

[1] Q maps the pair state-action into a real number.

state, regardless of what may happen next.
Stochastic policies can be learned more easily by using **Policy-Based** approaches, which has numerous additional benefits since we can learn arbitrary action probabilities, the agent is less reliant on the Markov assumption and performs better in partially observable situations [3].

2.2.4 Actor-Critic Reinforcement Learning

Combining the **Value-Based** and **Policy-Based** approaches is possible to define an **Actor-Critic** implementation. It is composed of the actor and critic network, where the first one decides the action that will be taken, and the second one communicates to the actor how good the action was and how it should adapt the actions in future iterations. Actor learning is based on the political gradient approach (*i.e.*, Policy-Based), while critic learning evaluates the action produced by the actor by calculating the value function as done in the Value-Based method. This RL approach will be seen in detail, from a mathematical point of view, in chapter 4 with the **Deep Deterministic Policy Gradient** technique that we will investigate in this thesis.

2.3 Mobile Robots and Reinforcement Learning

Modern autonomous mobile robot applications require efficient exploration of new environments as a primary prerequisite. The academic community is increasingly interested in combining robotics with reinforcement learning (RL) for designing reliable and efficient robotic exploration procedures that are appropriate for challenging real-world scenarios.
For achieving this purpose, RL attempts to simulate how people (or animals) learn through erroneous experiences in their environments. Therefore, the use of RL approaches in robotics is being studied in particular for the robot that autonomously learns to plan and regulate its actions in challenging and dynamic tasks. In real robotics applications, the agent has to deal with stochastic systems, among which: measurement noise, delays, the inability to accelerate the learning phase in the real-world environment, and safe exploration. Therefore, simulated environments are implemented to overcome issues defined previ-

ously. Although, this approach presents a crucial drawback, since the simulated environment model can quickly create and cause behaviours to diverge from what is expected due to the reducing real-world interaction. Several methods and approaches have been used to make the use of RL in real robotic applications realistic and to address the main issues, among which: finding a suitable reward shaping and the computational effort produced by the increasing spatial dimension (*e.g.,* discretizing the space and/or actions).
A reliable and effective exploration approach is required for the use of mobile robots in such rough environments. As a result, several exploration techniques (*e.g.,* artificial potential fields [5]) have been proposed in the past, which have allowed exploration for a variety of applications. However, in complex and dynamic real-world environments, it can be challenging to define exactly an efficient robust exploration method. Nowadays, exploration approaches integrate the hybrid concepts by planning future actions and responding to unanticipated events using information about the global environment. We can categorize the methods into two groups for simplifying the synthesis of how the RL algorithms typically construct the exploration strategies. However, information for both classes can be made up of processed sensor data like local maps, robot position, robot trajectories, or their combination [6]. The two classes are defined as follows:

- **end-to-end** approach [1], [7], [8], where the robot's exploration activities are performed as a black box. The inputs are passed into the RL algorithm, which returns the robot's control actions.

- **Two-stage** methods [9], [10], [11], can combine RL with non-learning-based techniques. However, RL algorithms can be used in either the first step or both. Therefore, the two stages are defined as follows:

    1. An algorithm, that takes as input the state, and determines the **target position** that must be reached by the robot.

    2. The **path planning** process is then performed on the agent from its current position to the target position.

3

Methods Design

In the following chapter, we are going to introduce the different methodologies, including the open-source robotics middleware system Robot Operating System also known as ROS, Gazebo, the mobile robot TIAGo and the machine learning libraries (*i.e.,* TensorFlow and Scikit-learn).

3.1 Robot Operating System (ROS)

Robot Operating System[1] or also known as ROS [12] is a middleware for the development of robot software, and it includes several libraries for robots' applications related to manipulation, navigation, and computer vision. It could be considered a meta-operating system and middleware. The reason is due to the properties such as hardware abstraction, low-level control devices and messaging among the process. The second one instead provides techniques for inter-process communications. Robot operating system is based on peer-to-peer (p2p) networks that are executed in such a way to process data among them. Moreover, we have different computation elements in ROS:

- Nodes: The nodes (Figure 3.1) are crucial for ROS implementations, that communicate using servers, services, or topics. The goal of nodes is to execute a process that accomplishes the computation.

[1]https://www.ros.org/

- Topics: The communication approach of ROS is a subset of publishers - subscribers pattern, and it permits to interchange of messages among nodes. The procedure is the following: nodes could subscribe to a topic and another node that has the function of a publisher can produce information that will be brought out and used by the subscribers. Notice that it is possible to carry out multiple publishers - subscribers toward a specific topic in such a way to enhance the exchange of information.

- Services: This chunk of the ROS system is constructed over request-replay interactions. This communicative implementation is an alternative way of communication that is not implementing the topics defined previously.

- Messages: A simple structure is defined for allowing an exchange of information, in a modular and standard way. It is possible to design custom messages using standard messages.

- Server's parameter: It runs inside the ROS Master, that will be explained subsequently, for storing and retrieving variables during the execution.

- Master: The ROS Master (Figure 3.1) provides naming and registration services to all present nodes and, in addition, tracks publishers - subscribers and service communication. The role of the Master is to allow individual nodes to locate each other. Therefore, when these nodes have computed their relative tasks, they can communicate among them through a peer-to-peer implementation.

- Bags: A bag is a file format in ROS for storing message data that plays an important role and a variety of tools have been written to allow to archive, process, analyze and view it.

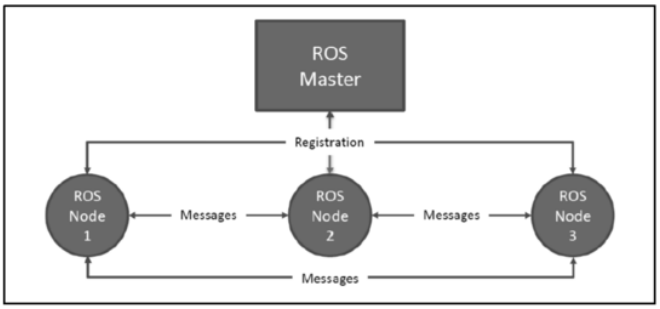


Figure 3.1: ROS's graph: The nodes are depicted as circles, the master with a rectangle and messages with the edges.

3.1.1 ROS Navigation Stack

Navigation Stack is the ROS package (Figure 3.2) that performs simultaneous localization and mapping (**SLAM**). SLAM is a fundamental technique that allows the robot to recognize the obstacles in its path and a route to navigate around them. This technique is a combination of multiple strategies that permit robots to navigate in situations that are partially known or completely unknown. **Location systems**, which determine its position with respect to a map and the surrounding environment, are another important element of the navigation stack. Adaptive Monte Carlo Localization (AMCL) is a technology that enables this through a static map. AMCL is a probabilistic location system for a 2D moving robot that uses a map and laser scans for generating pose estimations. **Gmapping** involves mapping an environment while the robot is moving. In other words, when the robot navigates around the area, it collects information from the surrounding environment using its sensors (*i.e.*, Lidar) and generates a map. In this way, robots can both create a map of an unknown region and update an existing map. For achieving this purpose, it takes to account the odometry information to generate more precise maps, since understanding the robot dynamics permits to estimate the pose. However, odometry information may be suggested to failure due to a lack of accuracy on capitation, friction, slip or drift. These errors accumulate over time make the data inconsistent and compromising the map formation [13].

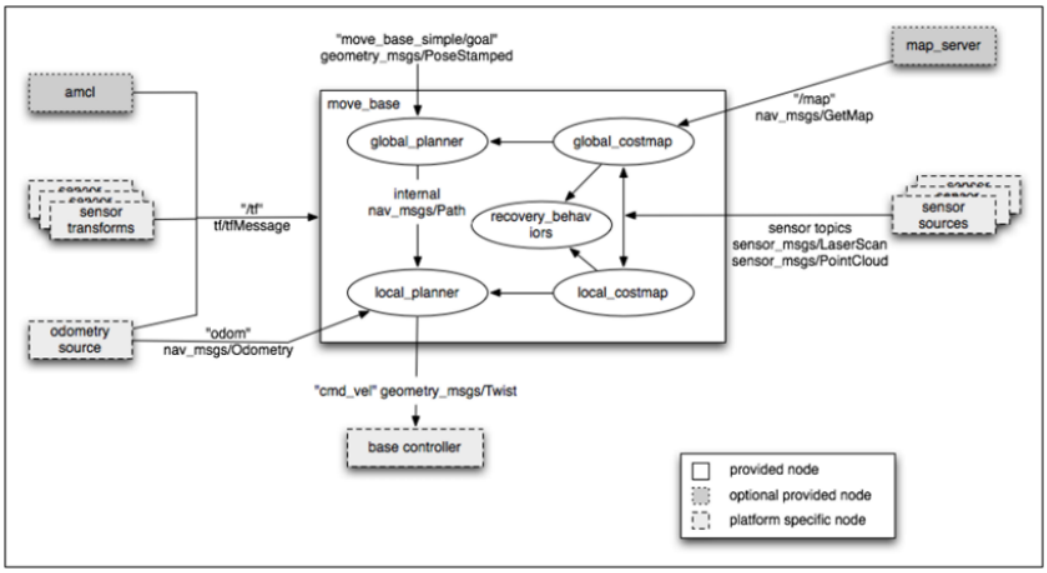


Figure 3.2: Overview of the navigation stack defined in [13].

3.2 Gazebo

Gazebo[2] is a 3D simulator, while ROS acts as an interface for the robot. The combination of the two creates a powerful robot simulator. With Gazebo, it is possible to create 3D scenarios with robots, obstacles and many other objects through physics engines for lighting, gravity, inertia, etc. A common use of this simulator is for testing different agents in challenging or dangerous scenarios without harming the robot. Therefore, most of the time it is faster to run a simulator than to run the whole scenario with your real robot. Gazebo was originally developed to test robot algorithms and verify different parameters for several applications, among which: error handling, battery life, location, and navigation.
Simulation experiments in this thesis have been conducted using Gazebo simulator and libraries that are supported by the GPU Nvidia GeForce RTX 3070Ti with a memory of 8 GB.

[2]https://gazebosim.org/

3.3 TIAGo

TIAGo[3] [14] is a collaborative robot manipulator that we will use in our experiments and thesis that is straightforwardly configurable and adaptable to different situations. To interact with the environment, it is equipped with different sensors (*e.g.*, 2D Lidar, Sonar and RGB-D camera) that are also applied for acquiring perception or mobility tasks. From a software point of view, PAL provides a simulated TIAGo model for Gazebo and also tutorials available that permit it to be performed with frameworks including ROS that we have seen previously. We are going to illustrate some technical specifications of TIAGo:

- BODY
  - Arm payload (at full extension) : 3 kg (without end-effector)
  - Arm reach : 87 cm (without end-effector)
  - Torso lift : 35 cm

- MOBILE BASE
  - Drive : Differential
  - Maximum speed : 1.5 m/s
  - Operational environment : Indoor

- SENSORS
  - Base : Laser 5.6 m / 10 m / 25 m range, rear sonars 3x1m range
  - IMU (Base) : 6 Degree of Freedom (DOF)
  - Motors : Actuators current feedback
  - Head : RGB-D camera

- COMPUTER

[3]https://pal-robotics.com/robots/tiago/

  - CPU Intel i5 / i7
  - RAM 8 GB / 16 GB
  - SSD 250 GB / 500 GB

- SOFTWARE
  - OS : Ubuntu LTS 64-bits, RT Preempt
  - Open-source middleware : ROS LTS
  - Arm control mode : Position / velocity / effort control

Figure 3.3: TIAGo's picture with its components [14].

3.4 TensorFlow

TensorFlow(TF)[4] is one of the most popular and powerful libraries for numerical computation, especially suited for large-scale Machine Learning (ML)[15]. It has been developed for providing a series of tested and optimized modules

[4]https://www.tensorflow.org/

for the creation of ML algorithms and models.
The calculus within a TF program are described through an acyclic graph, see Figure 3.4, where each node has zero or more inputs as well as outputs. The values flowing along the arcs represent arrays' tensor[5] of arbitral dimension. Ordinarily, the users build the computational graph using one of the supporting programming languages (C/C++ or Python) [16].
TensorFlow allows reusing the same model with different training data collection, as well as sharing to other users.

1. The model is built as a symbolic graph with placeholders for the input data and variables for the state of the model.

2. The program is run and optimized using input data provided by the user in the form of names.

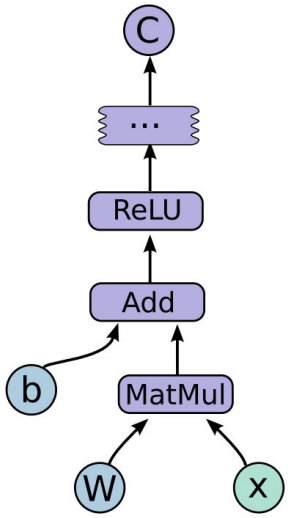


Figure 3.4: TensorFlow Dataflow Graph [16].

3.5 Scikit-learn

Scikit-learn[6] was originally called *scikits.learn* and was initially developed by David Cournapeau as a Google summer of code project in 2007. Later,

[5]**Tensors** are algebraic objects in mathematics and science fields that describe a (multilinear) relation between: sets of algebraic objects (related to a vector space).
[6]https://scikit-learn.org

in 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel, from FIRCA (French Institute for Research in Computer Science and Automation), took this project to another level and made the first public release (v0.1 beta).
Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modelling [17].
Key concepts and features include:

- Algorithmic decision-making methods, including:
  - **Classification**: identifying and categorizing data based on patterns.
  - **Regression**: predicting or projecting data values based on the average mean of existing and planned data.
  - **Clustering**: automatic grouping of similar data into datasets.
- Algorithms that support predictive analysis, ranging from simple linear regression to neural network pattern recognition.
- Interoperability with NumPy, Pandas, and Matplotlib libraries.

4 System Architecture

4.1 Shared Intelligence Approach

A **Shared intelligence** system was introduced in [2] for finding a trajectory through a fusion of multiple probability grids equally based. This is done without two main prior information: the surrounding environment and the final destination. Therefore, the concept of subgoal has been introduced for representing an intermediate destination in the surrounding area. For determining the subgoal position the system takes the maximum value of the fused policies (*e.g.*, user input) as represented in Figure 4.1.
Each policy is based on different information (*e.g.*, sensors) that handle multiple properties in the closeness space relative to the subgoal position. Those attributes are: the relative distance between the robot's position and the subgoal, the most probable direction, or the proximity of targets [2].
Now we are going to illustrate the policies implemented in the shared intelligence system proposed in [2]:

- The **obstacle-avoidance policy** is integrated for avoiding collisions. For reaching this aim, the robot has to sense distances for setting a secure and reachable subgoal represented by a probability value. Therefore, the policy is defined according to equation 4.1.

$$p_o = (1 - e^{-\lambda d_o}) \; with \; d_o > s_r \tag{4.1}$$

  Where s_r, d_o and $\lambda > 0$ are: size of the robot, the distance of the closest

obstacle (inside the area around the robot) and the smoothing parameter.

- The **distance policy** defines a range $[d_{min}, d_{max}]$ of possible values in which the subgoal may be defined. This is done for avoiding situations of conflict caused by the inability to move the robot from its current position.

$$p_d = ke^{-(d_r - d_{min})} \; with \; d_{min} \leq d_r \leq d_{max} \tag{4.2}$$

  Where, for values greater than d_{max}, Tukeys enclosure [18] is applied with $k = 1/(Q3 + 1.5IQR) = 1/(ln4 + 1.5ln3)/(d_{max} - d_{min})$.

- The **direction policy** is defined for guaranteeing the robot keeps uniform and smooth directions during the navigation phase, as represented in Figure 4.2. The policy is defined as follows:

$$p_d = \frac{1}{\sqrt{2\pi\sigma_d^2}} e^{\frac{-(\phi-\theta)^2}{2\sigma_d^2}} \tag{4.3}$$

  where the angle ϕ connects each position around the robot and the current robot's position and θ its robot's orientation.

- **user input** policy aims to increase the probability of reaching the subgoal by considering the command information given by the user. The current implementation is created with an attractive Gaussian probability distribution on the left and right sides, respectively defined with the equation 4.4:

$$p_u = e^{\frac{-(\phi-\theta)^2}{2\sigma_d^2}} \tag{4.4}$$

  Where d_u is computed as the distance between each cell (x, y) of the probability grid and a position u_i determined according to the users commands. In particular, u_i is placed at a distance d_{u_i} from robots position and rotated by 45° in the corresponding direction.

- The **fusion** of the policies is set for defining the subgoal. This is done by considering the major area of influence computed through a joint probability distribution of each policy according to the following equation:

$$\Sigma p = \prod_{k=1}^{N} p_k \tag{4.5}$$

Therefore, it is defined $\hat{\Sigma_p}$, that is a probability grid obtained by normalizing Σ_p. Then, for setting the subgoal position (x_{s_g}, y_{s_g}), the resulting grid is maximized. The position furthest from the robot is chosen for overcoming possible multiple maximum values (computed in equation 4.6).

$$(x_{s_g}, y_{s_g}) = \arg\max_{x,y} \hat{\Sigma_p} \tag{4.6}$$

The next step consists of finding the pose $(x_{s_g}, y_{s_g}, \sigma_{s_g})$. Where σ_{s_g} is set along the line connecting the current position and the selected position (x_{s_g}, y_{s_g}). Therefore, the pose is fed into the motion planner to identify the best way to reach the subgoal.

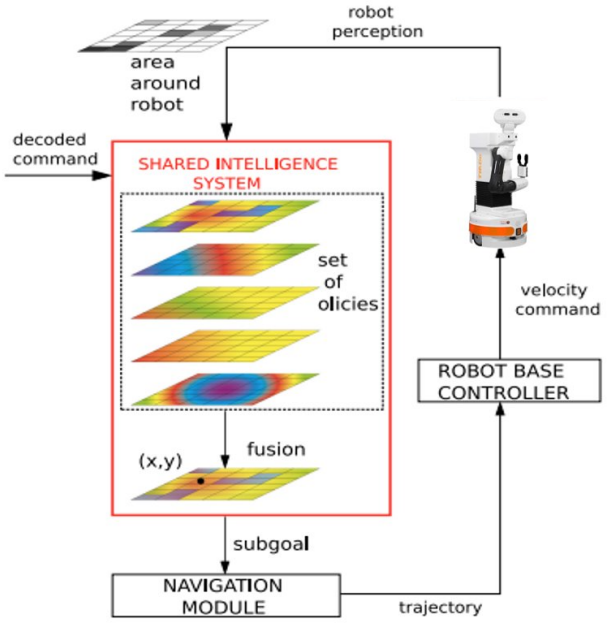


Figure 4.1: Schematic explanation of the policies that are equally merged to determine the robot's next subgoal, the figure is taken from [2].

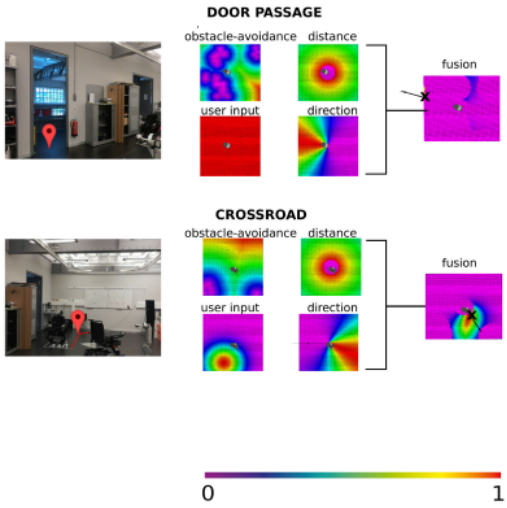


Figure 4.2: The figure illustrates various situations: the top view is a probability grid in which the red areas have the highest probability and the blue areas have the lowest. The computed subgoal (fusion section) is represented with a black arrow and a cross that represent the direction and position of the robot. The figure is taken from [2].

4.2 RL Policy

In this thesis, we extend the shared intelligence system proposed by Beraldo et al. [2] by adding a Reinforcement Learning policy (see Figure 4.4) that permits to understand the robot's behaviour through multiple trials and error iterations. Therefore, we hypothesize that the users navigation performance will increase with this policy.
The RL policy is defined according to equation 4.4, where instead of using the user information as input we implement the output of the model (*i.e.*, action) that will be presented in detail in the next sections. For achieving the objective described, we have defined an architecture (see Figure 4.3) that permits to train and test the RL model, and it could be described through the following elements:

1. The **environment** is based on Gazebo and ROS. It permits interaction with pre-processing module for defining the state.

2. The **pre-processing** phase analyses the occupancy grid and defines the **state** that is computed by applying clusters' algorithms on the local costmap and considering the closet points of the obstacles (respect to the robot's

position). It also defines a random goal inside the local costmap. We pay attention to choose a reachable goal that is not overlapped with an obstacle during the training phase. Although, this is not considered in the testing process. This is done for evaluating the robot's behaviour with a not reachable target.

3. The agent, which implements the **DDPG** algorithm, receives a state and computes the action (x and y position in the map) that represents the subgoal (*i.e.,* an intermediate step towards the goal) for the robot.

4. Given the action from the previous step, the robot tries to reach the goal position provided by the DDPG model according to the trajectory defined by the Navigation Stack.

5. Then, we have to deal with three possible situations:

   - **Collision**: The model proposed an action too close to an obstacle.
   - **Goal reached**: The goal is reached, then a new goal is set.
   - **Subgoal reached**: An intermediate position towards the goal (defined by the agent) has been successfully reached, so it will be estimated via **reward function** (only during the training phase) and then the DDPG will predict new actions based on the state defined in the preprocessing module.

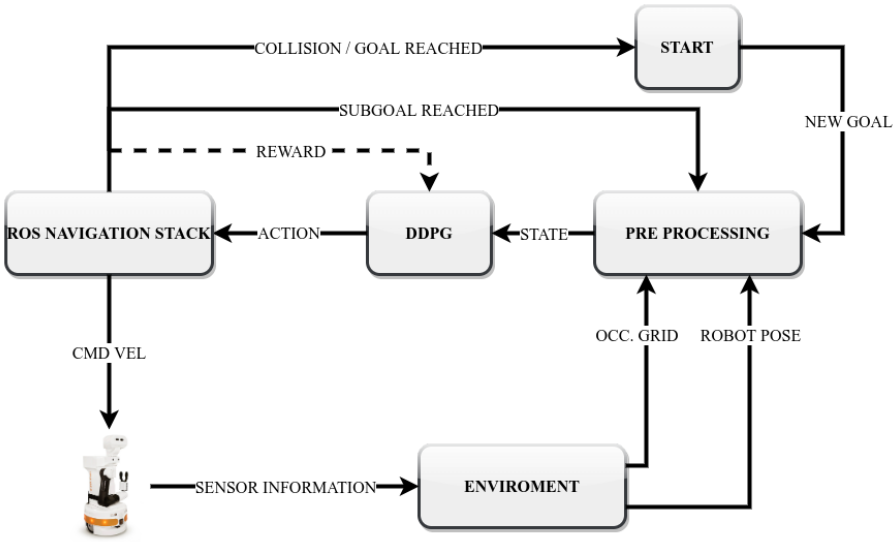


Figure 4.3: The proposed architecture implemented for training and testing the RL model.

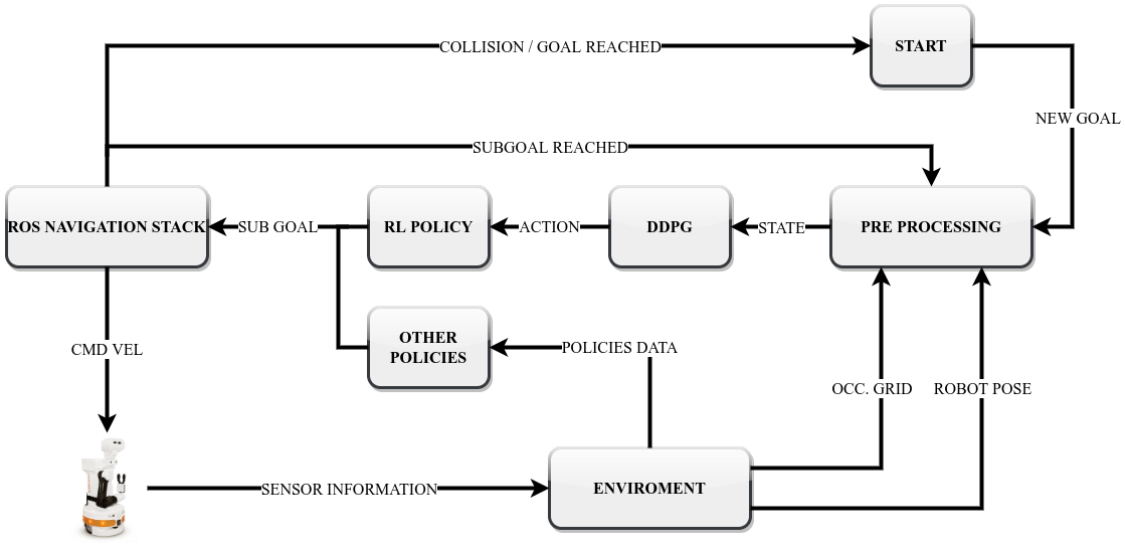


Figure 4.4: The proposed architecture that combines the current shared intelligence approach with the RL policy.

4.2.1 Cluster Analysis

Clustering is the process of organizing data into groups based on similar features within data points of the group. Similarly, this is applicable to other ML problems which show similarities in data. This is the aim of unsupervised learning, grouping a set of new data based on similarities among them.
In a **Hierarchical Classification**, the data are not partitioned into a particular number of classes or clusters at a single step. Instead, the classification consists of a series of partitions, which may run from a single cluster containing all individuals to n clusters each containing a single individual. Hierarchical clustering techniques may be subdivided into **agglomerative methods**, which proceed by a series of successive fusions of the n individuals into groups, and **divisive methods**, which separate the n individuals successively into finer groups [19].
In the next subsections, we will introduce different clustering algorithms that we examine in this thesis.

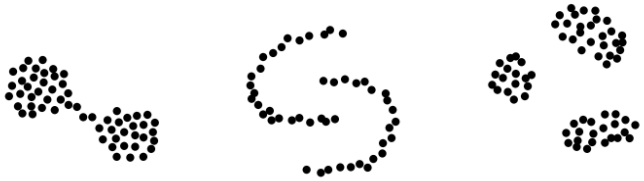


Figure 4.5: Clusters with internal cohesion and/or external solution. [19]

BIRCH Algorithm

BIRCH (Balanced Iterative Reducing and Clustering hierarchies) [20] is an advanced clustering algorithm useful for performing precise clustering on large datasets. BIRCH is a clustering algorithm that clusters the dataset first in small summaries, then those summaries are clustered. It is provided as an alternative method to *Mini batch KMeans* that converts data to a tree data structure, with the centroids being read off the leaf. These centroids may be the final cluster centroid or the input for other cluster algorithms like *Agglomerative_Clustering*.
BIRCH is a scalable clustering method based on hierarchy clustering and only requires a one-time scan of the dataset, making it fast for working with large datasets. This algorithm is based on the CF (clustering features) tree. In addi-

tion, this algorithm uses a tree-structured summary to create clusters where the structure of the given data is built by the BIRCH algorithm, called the clustering feature tree (CF tree). In the context of the CF tree, the algorithm compresses the data into the sets of CF nodes, those nodes have several sub-clusters known as CF subclusters that are situated in no-terminal CF nodes. The CF tree is a height-balanced tree that gathers and manages clustering features and holds necessary information of given data for further hierarchical clustering. This way, you can prevent working with whole data as input data. The tree cluster of data points as CF is represented by three numbers: N (number of items in subclusters), LS (vector sum of the data points) and SS (sum of the squared data points).
There are mainly four steps, which are implemented by the BIRCH algorithm [20]:

- **Scanning data into memory**: the algorithm scans the whole data and fits them into the CF trees.
- **Condense data** (optional): it resets and resizes the data for better fitting into the CF tree.
- **Global clustering**: it sends CF trees for clustering using existing clustering algorithms.
- **Refining clusters** (optional): it fixes the problem of CF trees where the same valued points are assigned to different leaf nodes.

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [21] is a clustering density based method that connects regions of points with high density. It is one of the most widely used clustering algorithms. DBSCAN estimates the density around each point (item) by counting the number of points in a neighbourhood (eps) specified by the user, and applies thresholds called $minPts$ to identify the core (all points with density values greater than a defined threshold), border (all points with density values lower than a defined threshold)

and noise points (all points that are not part of core or border set).
In the second step, the core points are collected in a cluster, if they are "density-reachable" (*i.e.* if there is a sequence of core points in which each of them falls within the eps-surrounding of the following element). Finally, the edge points are assigned to the clusters.
Then, for the DBSCAN clustering objectives, the points (items) are classified as core points, reachable or outliers, according to the following rules:

- A point p is a core point if at least $minPts$ points are within the eps distance. These points are directly reachable by p.

- A point q can be reached directly from p if the point q is within the distance eps from the point p (where p must be a core point).

- A point q is reachable from p if there exists a path $(p_1, ..., p_n)$ with $p_1 = p$ and $p_n = q$ where every p_{i+1} point is directly reachable from p.

- All points that cannot be reached from other points are outliers.

Now, if p is a core point, it forms a cluster with all other points that are reachable from it. Non-core points can be part of a cluster, but they cannot be used to reach other points.

Mean Shift

Mean Shift (MS) [22] is a clustering algorithm that aims to find blobs in a smooth density of samples. It is a centroid-based algorithm that works by updating possible candidates for centroids to be the mean of the points within a bandwidth. Therefore, to form the final set of centroids, it is important to remove duplicates (in the neighbourhood) through a post-processing phase in which the candidates are filtered.
One of the main advantages of the MS is the not dependency of the initialization since each point is considered a cluster. However, it is not highly scalable, as it requires multiple nearest neighbour searches during its execution that leads to

high computational time for solving the problem.
Therefore, we can summarize the main steps of the algorithm:

1. Initialization of the sliding window[1] and setting each data points as a cluster.

2. Each sliding window is moved towards higher density regions by shifting their centroid toward the data-points' mean until convergence.

3. Selection of sliding windows by eliminating overlapping windows. The overlap situation is caused by multiple windows moving to the same region.

4. Data points are assigned to the sliding window in which they are located at the end of the iterations.

4.2.2 Pre-Processing Module

One of the major stage for training a model of Reinforcement Learning is to define the **state**. In our case, it is composed of two information:

1. **Relative position** (x, y) between the robot position (received in input) and the goal defined at the beginning stage.

2. Set of **Distances** from the robot for tracing obstacles.

For defining those distances, several scientific articles use raw laser range sensors and then divide the whole laser scan into n evenly spaced circular sectors. Therefore, the minimum measured values for each sector are set as part of the state as proposed [23]. This kind of implementation may be subject to

[1] In the sliding window method, a window of specified length, moves over the data, sample by sample, and the statistic is computed over the data in the window.

noise and consequently imprecise measurements may cause a collision with an object. To overcome this possible issue, an occupancy grid is preferred since it combines the LIDAR information with other sensor data, among which: sonar and RGB camera positioned on the robot. Furthermore, it is already a key feature used by the system to compute other functionalities. Figure 4.6 shows an example of an occupancy grid where the pair $(80, 80)$ is the robot's position and each cell represents the probability ($[0, 1]$) whether is occupied or not.

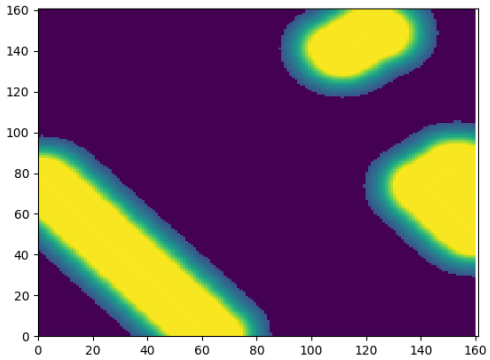


Figure 4.6: Example of an **occupancy grid map** representing the local costmap used in the robot's navigation. The zones in purple have a probability of being occupied by an obstacle close to zero, while the yellow one is more probable (~ 1).

Clustering the map was proposed in this thesis to group the objects with the same property. For this purpose, BIRCH algorithm was chosen and compared with other different implementations illustrated in the experimental, Chapter 5. Figure 4.7 and Figure 4.8 show how the distribution of obstacles are clustered. In particular, BIRCH algorithm sets data which is considerable as a single object into multiple obstacles, and it permits to the model to learn how to avoid huge obstacles knowing in advance multiple points (of the same object).
Therefore, for each cluster, the closest point is computed to abstract objects and simplify the computation of the distance between the agent and the surrounding obstacles.

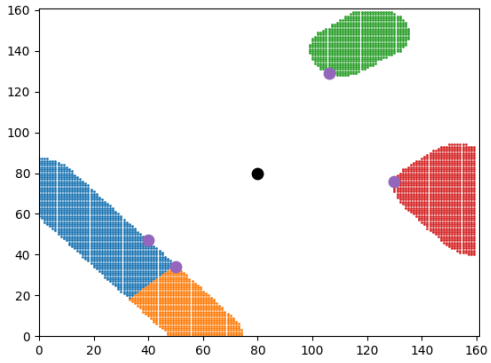


Figure 4.7: The cluster of the occupancy grid via **BIRCH** algorithm. The purple point indicates the closest distance from each detected cluster, and the black dot at the position $(80, 80)$ represents the robot's position.

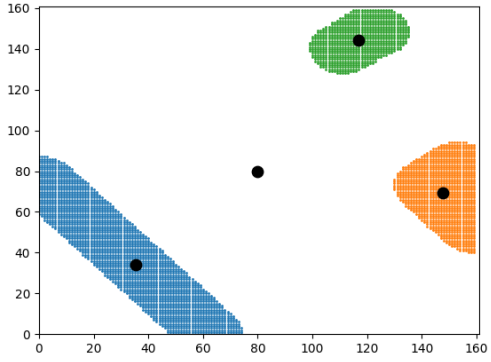


Figure 4.8: The cluster of the occupancy grid via **Mean Shift** algorithm. Where the black dot (on the single cluster) represents the relative centroid, and the black one at the position $(80, 80)$ represents the robot's position.

4.2.3 Deep Deterministic Policy Gradient (DDPG)

DDPG is an off-policy actor-critic model-free algorithm that combines Deep Q Learning (DQN) and DPG. DQN originally operated in discrete action space, but DPG extends it for continuous action space.
Deep Deterministic Policy Gradient (DDPG) is an algorithm (see Algorithm 1) which learns Q-function through the Bellman equation, and it uses the Q-function to acquire knowledge of the policy. Therefore, if we know the optimal action-value, $Q^*(s,a)$ (in any given state s) the optimal action $a^*(s)$ could be found using the following equation:

$$a^*(s) = \arg\max_a Q^*(s,a) \tag{4.7}$$

When the space action is discrete, finding the Q-values is trivial, since it is possible to compute it for each action separately, and therefore we can compare them. However, when the action space is continuous, we cannot comprehensively evaluate the space, so the optimization problem becomes a complex task. DDPG implements two approximators: $Q^*(s,a)$ and $a^*(s)$ in such a way to adapt the model for environments with continuous space action, it will be explained in detail below. An efficient gradient learning method rule can be set for a $\mu(s)$ policy assuming that the $Q^*(s,a)$ function is differentiable[2] with respect to the action argument a. Therefore, it is possible to approximate $\max_a Q(s,a) \approx Q(s,\mu(s))$ instead of running complex and expensive optimization operations [1].
Now, we will explain the mathematics parts of the DDPG algorithm, in particular for learning Q value and the policy.
We start by introducing **Q learning**, in particular with the Bellman equation (4.8) that describes the optimal action-value function $Q^*(s,a)$. $s' \sim P$ represents the sampled environment from a distribution $P(\cdot|(s,a))$ of the next state s'.

$$Q^*(s,a) = E_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right] \tag{4.8}$$

Suppose that the approximator is a neural network $Q_\phi(s,a)$ with variable ϕ and a set of collected transition (D) that are defined with a 5-tuple elements:

[2]Any function with one real variable that is differentiable has a derivative at each point in its domain.

(s, a, r, s', d). The first element (s) represent the current state, while a is the action, r is the reward function associated with the state and action, s' is the next state achieved with the action a and d is a variable that defines if the goal is reached. Given those assumptions, we can define a mean-squared Bellman error (MSBE) function that indicates to us the information of how much close Q_ϕ is to the Bellman equation (4.8).

$$L(\phi, D) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right] \tag{4.9}$$

All standard algorithms for training a deep neural network to approximate $Q^*(s, a)$ uses replay buffer[3]. This is the set D of previous experiences. To guarantee stable behaviour, the replay buffer should be large enough to contain a wide range of experiences, but it may not always keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning.
Q-learning algorithms rely on target networks. The term:

$$r + \gamma(1-d)\max_{a'} Q_\phi(s',a') \tag{4.10}$$

is called the target because when the algorithm minimizes the MSBE loss, it is trying to make the Q-function close to this target. Problematically, the target depends on the same parameters we are trying to train: ϕ that makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.
In DQN-based algorithms, the target network is just copied over from the main network every some-fixed number of steps. In the DDPG algorithm, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi \tag{4.11}$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1).
Putting together the assumptions seen previously, Q-learning in DDPG algorithm is performed by minimizing the flow based on loss function with a stochas-

[3]Number of experiences (*i.e.,* dataset) to store for training the model.

tic gradient descent (SGD):

$$L(\phi, D) = \underset{(s,a,r,s',d)\sim D}{E}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\right)\right)^2\right] \quad (4.12)$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

Policy learning aims to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s,a)$. Gradient ascent (with respect to policy parameters) is applied to solve equation 4.13 since the action space is continuous, and we assume the Q-function is differentiable with respect to action (*i.e.,* it is optimizable).

$$\max_\theta \underset{s\sim D}{\text{E}}\left[Q_\phi(s, \mu_\theta(s))\right] \quad (4.13)$$

Q-function parameters are treated as constants in equation 4.13 [1].

Algorithm 1 DDPG algorithm [1]

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^n.
Initialize target network Q' and μ' with weights $\theta^{Q\prime} \leftarrow \theta^Q, \theta^{\mu\prime} \leftarrow \theta^\mu$.
Initialize replay buffer R.
for $episode = 1, M$ **do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for $t = 1, T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$
    Execute action at and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
      $\nabla_{\theta\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|s_i$
    Update the target networks:
      $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for

Reward Function

One of the main goal of our navigation system is to reach a predefined destination while avoiding obstacles. The reward function was developed by taking inspiration from the APF formulation, and it is thus mainly influenced by two contributions, defining the attracting and the repelling potential field. To calculate the attractive component, the system relies on a localization algorithm that provides the position of the robot in the world reference frame. Based on this information and using equation 4.14, the attractive force is directly proportional to the Euclidean distance between the position of the robot p_r and the target p_g in the current time step:

$$U_{att} = \alpha \|p_g - p_r\|_2 \tag{4.14}$$

where α is a positive gain that has been empirically defined.
For calculating the repulsive potential contribution, the obstacles generate a repulsive force on the robot(4.15). For this purpose, we use the cluster approaches described in the previous section. Therefore, we compute the repulsive contribution as follows:

$$U_{rep} = \beta \sum_{i=1}^{N} \left(\frac{1}{k + l_i} - \frac{1}{k + l_{max}} \right) \tag{4.15}$$

- β is the positive gain defined empirically. It is defined according to equation 4.16.

$$\beta = \begin{cases} \delta & \text{, if } \|p_g - p_r\|_2 > d_{infl} \\ \frac{\delta}{exp[4(d_{infl} - p_{goal})]} & \text{, otherwise} \end{cases} \tag{4.16}$$

  where δ is a positive gain and $d_{infl} = 0.75 l_{max}$.

- $k(= 0.8)$ is a constant used to limit the repulsive field.

- N the number of obstacles detected in current time t

- l_i is minimum (Euclidean) distances towards the obstacle i.

- l_{max} is the maximum value for avoiding zero division in equation 4.15.

The reward function is computed according to equation 4.18 from the two attractive and repulsive components as proposed in [23].

$$sh_t = -U_{att} - U_{rep} \tag{4.17}$$

$$reward_t = sh_t - sh_{t-1} \tag{4.18}$$

The reward function $reward_t$ considers the value of the function sh at time t and $t-1$, by adding, a sort of memory in the system.

Model definition

The architecture of the DDPG network explored in this thesis, shown in Figure 4.9, includes:

1. The state (computed during the pre-processing phase) given as input to the **Actor** that returns the action that the robot must implement.

2. The pair state and action are taken as input by the **Critic** network and return a Q value that will be used for minimizing the loss function defined by algorithm 1.

The number of hidden layers set is defined in Chapter 5 with the relative performance.

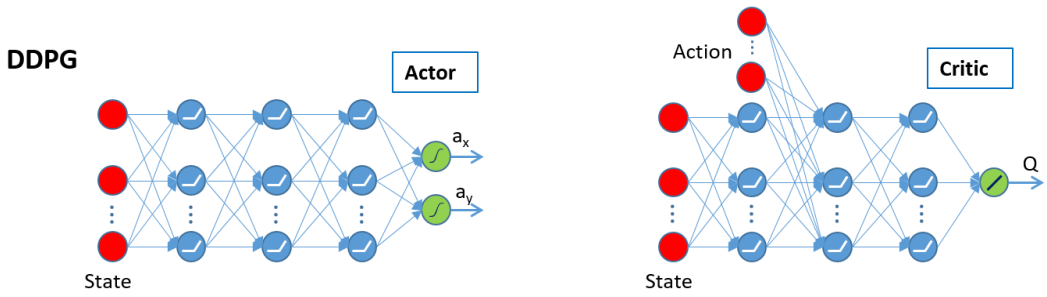


Figure 4.9: DDPG model divided in actor and critic.

The model, as reported in the Figure 4.9, is characterized by the following parameters:

- γ **reward** value that is the gain for avoiding the obstacles.

- **collision reward**, that is, the penalty used in case of collision between the robot and the objects.
- α **reward** coefficient for weighting the importance of reaching the goal.
- **state** that considers 8 or 14 elements of a vector that represents the position (of the robot) and the six closest obstacles. More details are provided in the next section.
- **batch normalization**, that is a method used to make the model faster and more stable through normalization of the layers' inputs by recentring and rescaling. We have considered some models with this implementation and others not.
- **configurations** of the hidden layers, changing the number of neurons and consequently the parameters to learn. The values are reported in the experimental section.

However, some parameters are fixed, chosen according to [23] and [1]:

- **batch size**(= 64), that defines the number of samples that will be propagated through the network.
- **max buffer**(= 50000), number of "experiences" to store. This constant variable is important since a value too low leads to a bad generalization. Conversely, too high a value can slow down the training.
- **goal reached** (= 10) is a value that is given as a reward when the robot reaches the goal.
- **max number of iterations** (= 70), the value set an upper bound for the number of possible actions for each epoch.
- **standard deviation of the OU noise** (= 0.008), set to facilitate the robot exploration.
- **gamma** (= 0.99), discount factor. (Always between 0 and 1)
- **tau** (= 0.001), this coefficient for updating the target parameters.
- **upper (** $= \pm 0.25$**)** and **lower (** $= \pm 0.02$**)** bound for each action (x, y). This is done for making small improvements toward the goal.

- actor's and critic's **learning rate** (= 0.001) that determines the step size at each iteration while moving toward a minimum of the loss function.
- *Adam* has been chosen as a **optimizer** that is an update of the *RMSProp* optimizer, running averages of both the gradients and the second moments of the gradients are used.
- The activation functions in the model are defined as follows:
  - **ReLu** for hidden layers, it is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. All the hidden layers are initialized with *HeNormal* algorithm, since it is the most appropriate initializer for *ReLu* functions. In this method, the weights are initialized keeping in mind the size of the previous layer, which helps in attaining a global minimum of the cost function faster and more efficiently. The weights are still random but differ in range depending on the size of the previous layer of neurons. This provides a controlled initialization, hence the faster and more efficient gradient descent.
  - While function **Tanh** takes any real value as input and outputs values in the range -1 to 1. This is ideal for controlling the robot in all directions.

4.2.4 Training Process

The simulation environment used for training the agent is shown in Figure 4.10 and such an environment simulates the IAS-Lab laboratory scenario at the Department of Information Engineering, a typical office environment with desks, and chairs as obstacles, strict passages and open spaces. To train the agent, the simulation was configured in an episodic setting, where each episode consists of a sequence of steps (1250 *epochs*). At the beginning of each episode, the robot and the goal are placed in a random location of the scenario, as it could be seen in Figure 4.11. This choice helps the robot to better explore space and learn how to reach different goals. At each time step during the

training process, the agent performs an action with added noise according to an **Ornstein-Uhlenbeck** (OU) distribution and receives a reward given by equation 4.18 seen previously.
The distribution is defined according to the equation 4.19.

$$dx_t = -\Theta x_t d_t + \sigma dW_t \tag{4.19}$$

where $\theta > 0$ and $\sigma > 0$ are parameters and W_t denotes the Wiener process [24]. Since the policy is deterministic, if the agent explored spatial action, they would initially not try enough actions to find useful learning. To make the DDPG policy more explorable, we add noise to its actions at training time. The authors of the original DDPG paper [1] recommended time-correlated OU noise, but recent results suggest that uncorrelated Gaussian noise with a mean of zero works very well. To get higher quality training data, it is possible to reduce the amount of noise as the training progresses.At test time, to see how well the strategy exploits what it has learned, we do not add noise to the actions (during the tests) for avoiding the creation of bias and to assess the learned navigation behaviours.
This process is repeated until the maximum number of steps, 70 in our case, is reached or when the robot moves into a final state that occurs when it collides with an obstacle or when it reaches the goal. In the first case, the agent receives a negative reward, while in the second situation, it receives a positive reward.
During the training process of the actor and the critic neural networks, the Adam Optimizer was used with a base learning rate and a minibatch size. For selecting the most suitable agent, we perform the comparison between different configurations of the DDPG agent, by increasing the number of neurons in each hidden layer (actor and critic), changing the hyperparameters of the equations 4.15 and 4.14 and the state definition. We have considered two kinds of possible states: the distances from the obstacles defined as an absolute value or defined as 2D value (x, y). In both cases, they are relative distances between the closest point of each cluster and the robot's position. In addition to this, we have also to take in consideration the relative position of the robot with respect to the sub-goal $(rel\_pos\_x,\ rel\_pos\_y)$. Therefore, in the first case a vector state of 8 elements $(rel\_pos\_x,\ rel\_pos\_y,\ distances * 6)$ has been defined, while in second situation a vector of 14 elements is set as $(rel\_pos\_x,\ rel\_pos\_y,\ distances\_x * 6,\ distances\_y * 6)$.

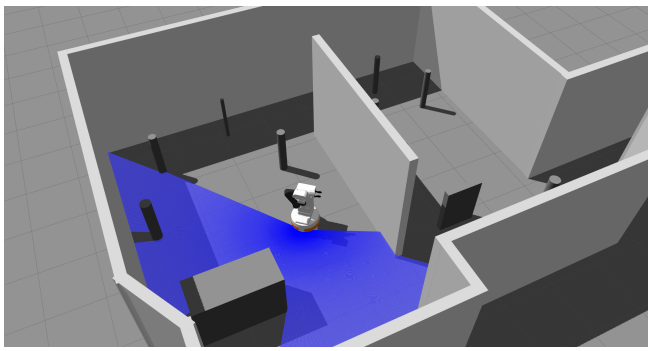


Figure 4.10: Gazebo-based simulation scenario used for training the RL system in this thesis.

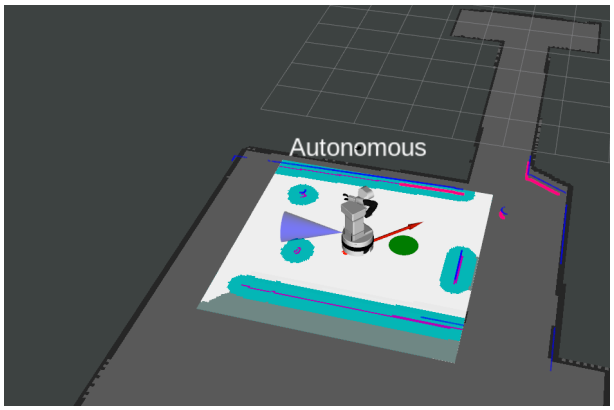


Figure 4.11: Scenario in which the robot is planning a path to reach the current goal represented with the green circle. For achieving this result, subgoal per time (red arrow) is given. The highlighted square area is the local cost map containing the obstacles and walls (blue zones) around the robot.

5 Evaluations

In this chapter, we will explain and show the results obtained during the training and testing the model in different conditions. We also compare different cluster algorithms for the pre-processing module. Finally, we report the performance of the overall system, combining the proposed RL policy with those defined in [2].

5.1 Clusters Performance

For determining the most appropriate algorithm for clustering. We have performed a pilot experiment where we compare the three algorithms described in Section 4.2.1, the **DBSCAN**, **Mean Shift** and **BIRCH**, according to the execution time. We have analysed the results 18 maps (similar to Figure 4.6) for estimating which algorithm is more suitable. Each map element has a $(0, 1)$ probability of occupancy associated with it, we applied two different thresholds, 0.30 and 0.95, in such a way to consider only the cells with higher probability (*i.e.,* reducing the shape of the obstacle) and consequently to simplify the calculation time.
The results are shown in Figures 5.1 and 5.2. DBSCAN is the most common algorithm for clustering, and it is quite fast, but it is unable to divide the map into a more homogeneous form as we have discussed in section 4.2.4. This is done by the Mean Shift algorithm, however, it is computationally expensive for a real-time application. For this reason, in this thesis, we have chosen to apply the BIRCH algorithm that combines those two characteristics.

With this purpose, we need to determine the obstacles' position. For achieving this aim, we have tested two options: The first one is the **alpha shape** [25], or α-shape, that is a family of piecewise linear simple curves in the Euclidean plane associated with the shape of a finite set of points. Once applied *alpha_shape*, we have to compute the distance from the shape obtained and the robot's position. The second is a simple calculation of the closest point of each cluster found, comparing the distance of each of them and choosing the minimum one. The experimental results, shown in Figures 5.1 and 5.2, revealed that BIRCH algorithms with the classical closest point computation perform as the best approach. We have set the probability threshold to 0.30 since the difference (from the computational point of view) with the other one chosen (*i.e.*, 0.95) is minimal with BIRCH algorithm.

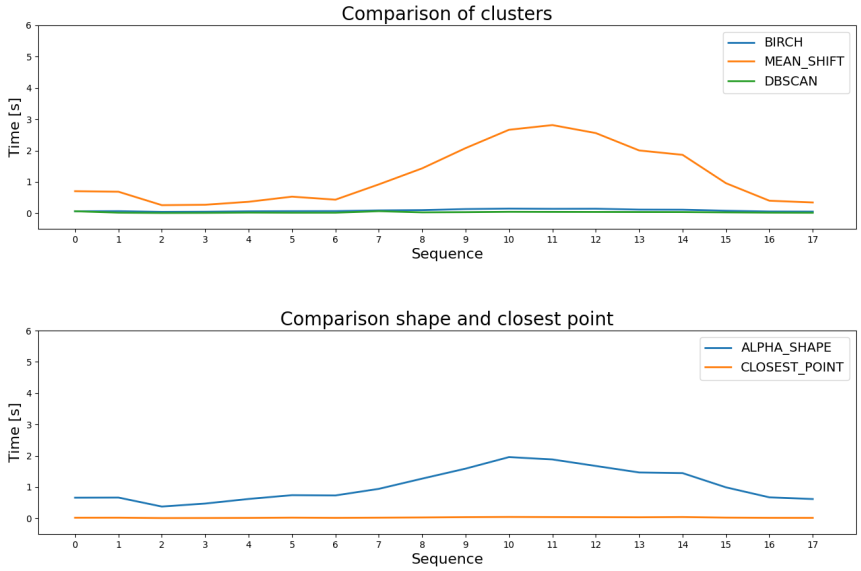


Figure 5.1: Comparison of the clusters and closest points with $threshold = 0.3$.

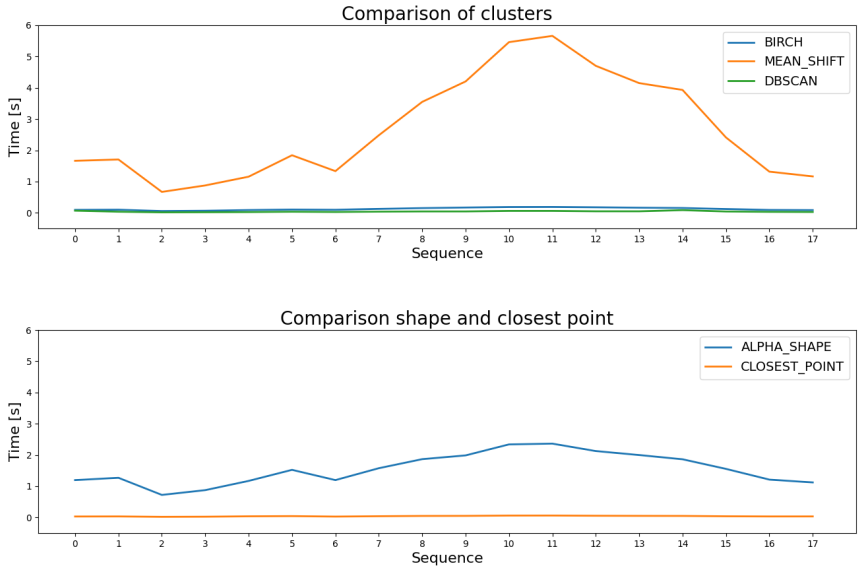


Figure 5.2: Comparison of the clusters and closest points with $threshold = 0.95$.

5.2 Training Phase

In the previous chapter, we have defined the training process and the simulation environment (Figure 4.10) with different obstacles. Now, we are going to illustrate the results achieved during the training phase.
Furthermore, Figures 4.11 and 5.3 show two trends for each model. The first one is computed considering an average of 200 episodes, while the second one is the average of 40 episodes (lighter colour dotted trend).

5.2.1 Reinforcement Learning Model Comparison

We initially tested a configuration implemented in [1] considering a low value of $gamma$ (γ). The configurations reported in Table 5.1 have some issues since during an initial testing period they perform badly and different collisions have been detected. This fact is due to low $collision$ and $gamma$ (γ) values set into the reward function. Therefore, as reported in Table 5.2, the negative actions are weighted with a higher value.
We have also taken in examination the efficiency of applying $Batch\ Normalization$ (BN) in this specific context. As shown in Figure 5.3 the model has a faster convergence with the BN implementation and for this reason, it has been applied in all trials reported in Table 5.2.

Training models 1					
Agent	BN	Coll. Rew.	γ	Actor hidden	Critic hidden
1	False	-100	2	400x400x400	400x400x400
2	True	-100	2	400x400x400	400x400x400
3	True	-100	8	400x400x400	400x400x400
4	False	-150	4	400x400x400	400x400x400
5	False	-185	4	600x600x600	400x400x400
6	False	-185	8	600x600x600	400x400x400
7	False	-185	4	400x400x400	600x600x600
8	True	-185	4	400x400x400	600x600x600
9	True	-200	4	800x600x400	800x600x400

Table 5.1: Trainig model with $state$ composed of 8 elements and α reward is setted as 100.

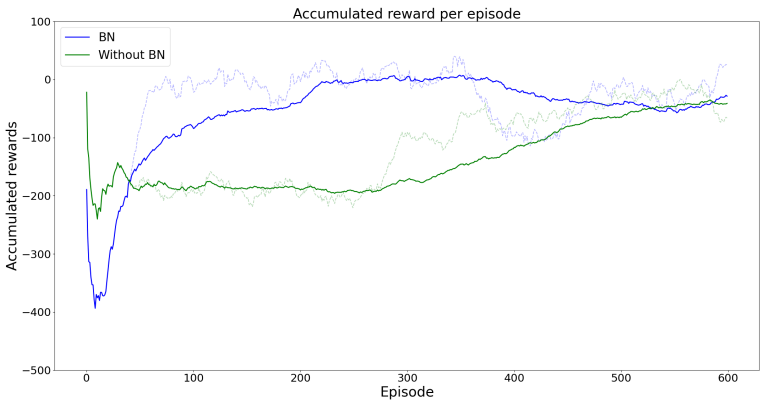


Figure 5.3: Comparison of two identical models where the BN implements the $Batch\ Normalization$ and $Without\ BN$ not.

Changing the reward function parameters by reducing the α value and increasing the γ the number of collisions should be remarkably reduced. In other words, we are trying to maintain the robot as far as possible from obstacles and at the same time, reduce the gain of attractive field. In this way, the robot should learn to avoid goals that are not safe. For this reason, we have decided to continue testing (that will be explained in the next section) only the models

defined in Table 5.2.

Training models 2						
Agent	BN	Coll. Rew.	γ	α	Actor hidden	Critic hidden
1	True	-200	20	50	600x600x600	400x400x400
2	True	-200	16	60	800x600x400	600x600x600
3	True	-200	18	70	600x600x600	600x600x600
4	True	-200	18	70	400x400x400	600x600x600

Table 5.2: Trainig model with $state$ composed of 14 elements.

We have reported the Q and reward accumulated values of those methods that have achieved better performance and better generalizes during the training phase that will be tested in static and dynamic conditions. As represented in Figure 5.4, all methods in Table 5.2 have been able to converge well, in particular method 1 and 4.

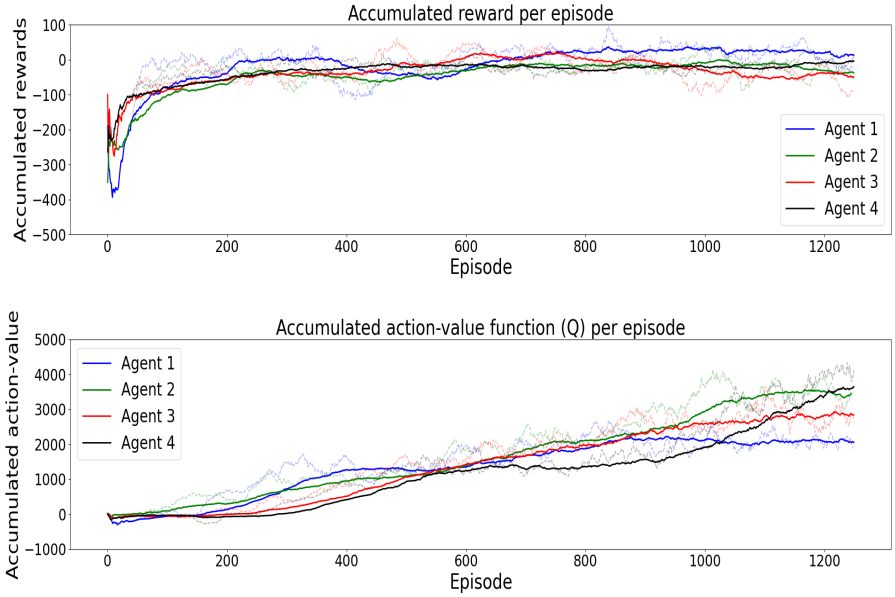


Figure 5.4: (Top) Accumulated reward per episode achieved with the agents reported in Table 5.2. (Bottom) The accumulated action-value function (Q) per episode achieved with the agents reported in Table 5.2.

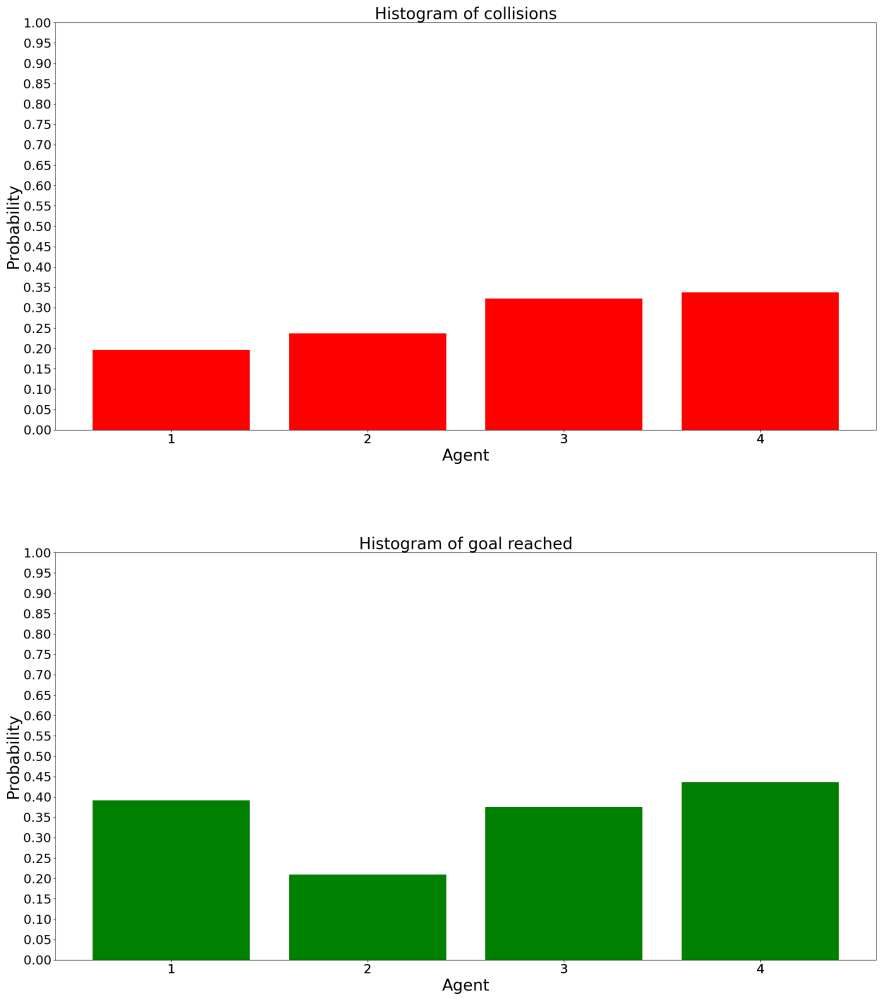


Figure 5.5: Testing results of the most relevant methods. Top: it represents the ratio between the number of collisions and the epochs. Bottom: The ratio between the goal reached over the epochs by the robot. There are some situations in which the number of maximum iterations is reached, or the goal is too close to the objects and consequently is not reachable in a safety way.

5.3 Testing Phase Results

During the training phase, we have found different models (Table 5.2) that perform well. Therefore, in this section, we will report the results of the test phase of those models in a different simulated environment to check if the model generalize properly.
We have evaluated the selected four models in two scenarios: tests with static (an example is shown in Figure 5.6) and dynamic (an example is shown in Figure 5.7) obstacles. We have tested each model for 25 trials, and we compare them according to the following metrics:

- **Safety Accuracy (SA)**, this value, represents the probability of non-collision during the tests. This is the most relevant metric since we can understand if the system is safe enough to be tested in the real world. There are several situations (Figure 5.8) where the target is not reachable or is too close to an obstacle, and must be avoided for security reasons.

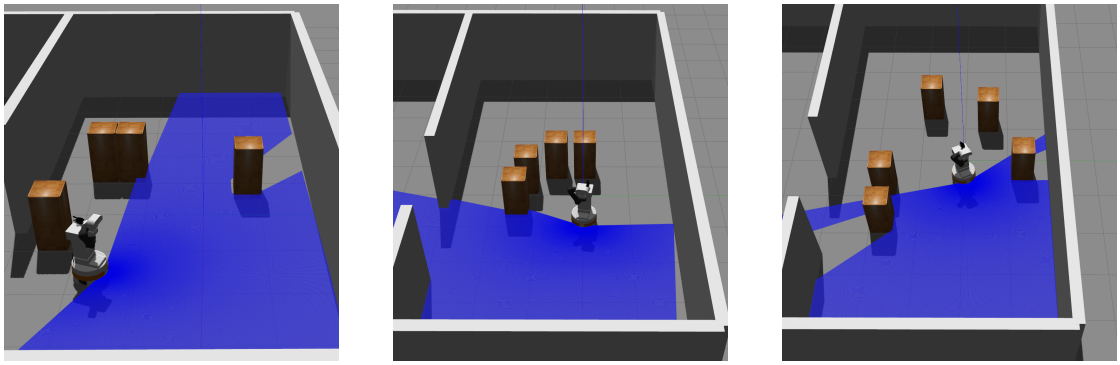
- **Goal Accuracy (GA)** represents the number of goals reached over the total number of tests tried.

- **Average Path Length (APL)** measures the typical path length, estimated in meters.

- **Average Time (AT)** for reaching a feasible goal.

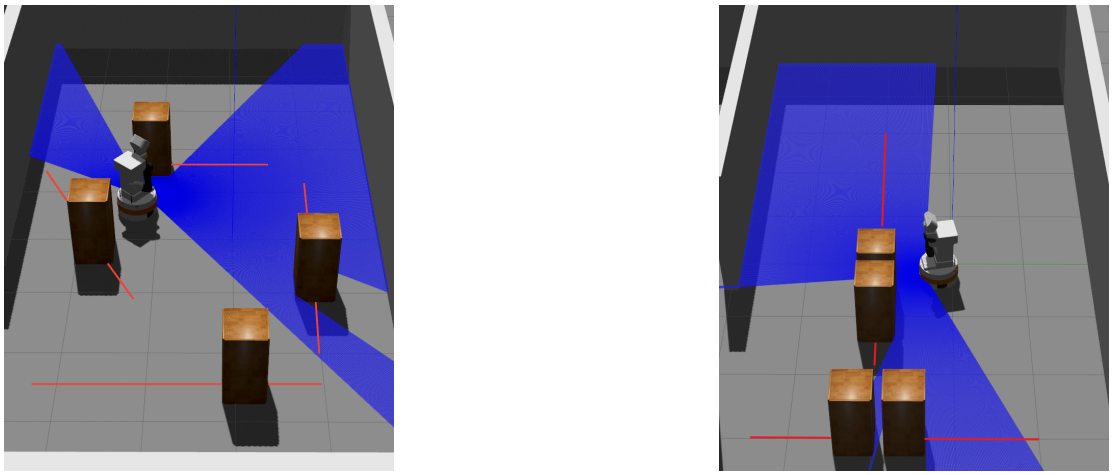
- **Mean Distance (MD)** provides the measurement between the robot and the obstacles during the comparison test.

The goal is defined randomly inside the occupancy grid. In this testing phase, it is possible that the goal is set on an obstacle (such as Figure 5.8). The reason for this choice is to understand if the robot will try to reach the goal regardless of whether it is an unreachable goal.



(a) Environment 1 (b) Environment 2 (c) Environment 3

Figure 5.6: Static Gazebo environments



(a) Environment 1 (b) Environment 2

Figure 5.7: Dynamic Gazebo environments where the red arrows represent the obstacles' path.

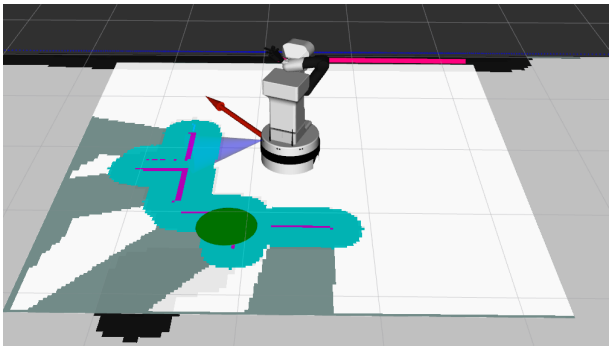


Figure 5.8: Particular case of goal set on obstacles during test phase.

5.3.1 Testing With Statics Obstacles

The first step for evaluating the performance of the agent is to change the environment (i.e., alter the positions of the obstacles around the robot) and analyse its behaviour. For achieving this purpose, we have considered three different situations (represented in Figure 5.6). The first one (Figure 5.6a) is the most general situation in which we have four objects that are positioned not too close to the robot. The second one (Figure 5.6b) is defined as investigating the behaviour of the robot with huge obstacles around it and how it can avoid them through a sequence of safety actions. The last one (Figure 5.6c) is the most challenging for the robot since it is surrounded by all the obstacles and the computation of feasible paths for avoiding them may require more time as reported in Table 5.5.

Static environment 1					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	40	96	0.9311	0.4174	10.1381
2	32	80	1.4221	0.3476	16.2131
3	28	96	1.9037	0.3622	29.5850
4	**48**	**100**	1.9497	0.3413	25.0579

Table 5.3: Test results of the examined models in Figure 5.6a

Static environment 2					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	36	84	1.7853	0.2893	27.8005
2	32	80	1.1002	0.2224	13.1312
3	40	**96**	1.9037	1.6319	28.8072
4	**60**	92	2.2127	0.2791	23.5084

Table 5.4: Test results of the examined models in Figure 5.6b.

Static environment 3					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	20	**92**	2.3757	0.3264	28.1148
2	**40**	68	1.1182	0.2763	21.9120
3	20	84	2.5398	0.2618	47.9996
4	36	80	1.4603	0.2969	22.5144

Table 5.5: Test results of the examined models in Figure 5.6c.

5.3.2 Testing With Dynamics Obstacles

The second step consists of testing the four models in more complex and dynamic environments. For this purpose, we have introduced dynamic obstacles (Figure 5.7) in the environment. Such obstacles move along a predefined path with a constant velocity. For testing those environments, the frequency rate of updating the local cost map has been increased to perceive possible objects as soon as possible. We have considered two possible situations: the first aims to prove the efficiency when the agent is surrounded by dynamic obstacles, while in the second situation the robot has to face large dynamic objects.

Dynamic environment 1					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	40	**80**	1.3497	0.2216	20.0835
2	48	68	1.6920	0.2320	34.0909
3	44	68	1.5366	0.2845	23.9742
4	**64**	76	1.3567	0.2462	19.7002

Table 5.6: Test results of the examined models in Figure 5.7a.

Dynamic environment 2					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	20	**88**	2.4850	0.3180	29.9104
2	32	**88**	1.6830	0.3910	30.6781
3	**44**	72	0.8893	0.4017	7.7372
4	28	**88**	2.0603	0.3200	30.7482

Table 5.7: Test results of the examined models in Figure 5.7b.

5.3.3 Summary Evaluation of the Examined Models

In this section, we discuss which model could be more suitable for this thesis. For comparing the different approaches, we apply a weighted average among the different tests. In particular, we assign a higher value to the results achieved in the dynamic scenarios, since it is more challenging for the agent. Therefore, we report the equation implemented to assess the results, and we apply that for each metric:

$$result = \frac{\sum_{i=1}^{5} w_{env} m_i}{\sum_{i=1}^{5} w_{env}} \tag{5.1}$$

where m is the metric considered and w_{env} is set equal to 1 in case of static environment and equal to 1.5 for dynamic case.
The results are summarized in the Table 5.8 which shows that the highest performance is model 4 considering the robot's ability in achieving the goal. This result depends on the definition of the reward function parameters (Table 5.2) since the attractive field was set with a larger value than other methods. Although, the more safe agent (1) is the one that has γ parameters higher and consequently repulsive field greater compared to other implementations. Considering the models 3 and 4, that share the same parameters of the reward function, we can infer that increasing the number of hidden layers (as done in model 3) cannot enhance the quality of the performance. APL columns establish that model 3 has an average path length shorter compared with the other models. In other words, it means that the robot is able to reach only the goal that is close to the current position, and this is one of the major reasons why it has not been chosen as the best estimator. The consequence of a higher repulsive field coefficient is that the robot keeps a farther distance from the obstacles. Therefore, the mean

distance from obstacles (MD) with the higher value is strictly correlated with the Safety accuracy (SA) parameters and this is proven since in both cases model 1 performs well. As regards the time (AT), there are any essential differences between the models.
As stated earlier, we chose model 4 as the best estimator. Indeed, it can be considered a safe implementation since its results are close to the performance of model 1 regarding the SA metric, and it achieves goals with greater accuracy.

Weighted average of the models in all enviroments					
Agent	GA [%]	SA [%]	APL [m]	MD [m]	AT [s]
1	31	**87.3**	1.8073	**0.3071**	**23.5073**
2	37.3	77	**1.4505**	0.2968	24.5683
3	36.6	81	1.6191	0.2845	25.6598
4	**47**	86.3	1.7913	0.2944	24.4589

Table 5.8: Summary of the general performance of the four analysed models.

5.4 Evaluation of the New Shared Intelligence Approach

In this section, we will present the evaluation of the state-of-the-art shared intelligence approach defined in [2] combined with the Reinforcement Learning policy. We have considered four different RL policies which are defined according to the four best models tested in the previous section. For a fair comparison, we also evaluated the case where reinforcement learning is not applied. This is done to understand the performance improvement of the new approaches.
Environmental setup shown in Figure 5.9 is composed of multiple obstacles (*e.g.*, tables), some of them are dynamics as we have seen in the previous section.
Each system is tested 3 times. For each test, we have defined 7 target positions that the robot has to reach to define the expected robot's path. The user's commands can be left or right and are delivered by the user via the keyboard. Such commands are then managed by the user's policy. We have also considered distance, direction and obstacle-avoidance policies. The last one is redundant, however, it is necessary for avoiding the wrong action predicted by the model.

In some particular cases, the robot may collide with an obstacle or the number of iterations for achieving the target positions are numerous, in these situations the test is restarted from the closet target positions and counted as a failure in the accuracy.
We have re-defined and added new metrics for evaluating this test properly. The analysed metrics are the following:

- **Goal Accuracy Policy (GA_P)** represents the number of target positions reached over the total number of trials.
- **Average Time Policy (AT_P)** for completing the entire path (all 7 target positions).
- **Average Number of User's Input (ANUI)** for completing the entire path.

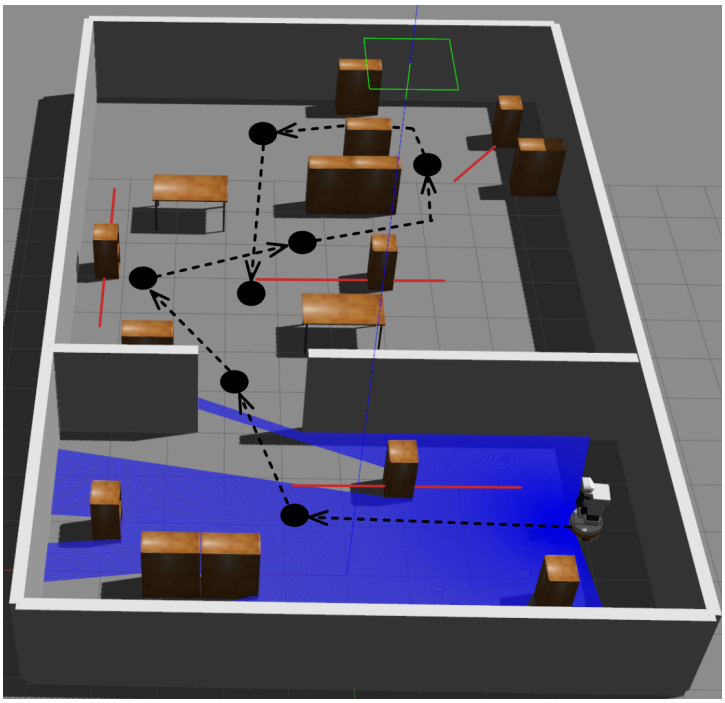


Figure 5.9: Experimental setup to evaluate the new shared intelligence system. We include dynamic and static obstacles. The red lines represent the objects' path while the black circles the target positions to reach.

5.4.1 Results Of New Shared Intelligence Approach

In this part of the thesis we evaluate the final results achieved with the new proposed approach. The results are described in Table 5.9 which compares the

examined conditions according to metrics defined in the previous sections.
Two approaches with better performance according to the navigation accuracy (goal reached) and number of collisions are the original system (SI) and the Shared Intelligence Reinforcement Learning Policy based on model 4 (SI + $RLP4$) implementation, which is also the most performant model estimated during an initial testing phase (Section 5.3.3). From APL metric, it is worth noticing that (SI + $RLP4$) performs significantly worst compared with other due to high γ value that avoid goals in the proximity of obstacles. The mean distance from obstacles (MD) shows, as before, that the model Without RL works fine, and it maintains a route that is further from obstacles. However, this could be a drawback since the robot cannot choose a path close to the obstacles even if it is shorter compared to the one estimated by the current shared intelligence implementation and or docking close to them. Therefore, it is necessary to combine a good trade-off between the MD and AT_P. In other words, the approach (SI + $RLP4$) combines a safe approach with the average time for completing the overall path. Consequently, user's input commands necessary are reduced, as reported in the column $ANUI$ of the Table 5.9.
In the conclusion, The Shared Intelligence + RL Policy Model 4 guarantees a safe implementation as the original system. However, it required a reduced time necessary for completing the path, the number of human interventions and navigation accuracy is increased. We have also reported the two trajectories (see Figure 5.10) computed by the two best systems (SI + $RLP4$ and SI).

Evaluation Of Shared Intelligence Approaches						
System	GA_P [%]	SA [%]	APL [m]	MD [m]	AT_P [s]	ANUI
SI+RLP1	76.19	85.71	43.1800	0.0658	1077.853	123
SI+RLP2	80.95	85.71	37.3314	0.0635	735.7410	119
SI+RLP3	85.71	95.23	39.3657	0.0615	830.5356	85
SI+RLP4	**100**	**100**	38.9604	0.0689	**571.7233**	**63**
SI	90.48	**100**	**36.8820**	**0.0787**	809.5110	84

Table 5.9: Test results of the examined shared intelligence systems in the setup illustrated in Figure 5.9.

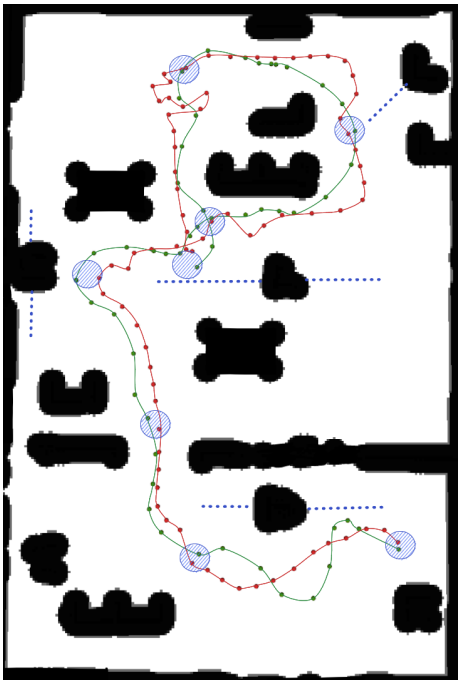


Figure 5.10: Trajectories of $SI + RLP4$ (green path) and SI (red path) approaches. The blue circles represent the target positions while the blue dotted lines are the dynamic obstacles. The black elements are the obstacles or walls sensed by the robot during the navigation.

6

Conclusions and Future Works

In this thesis, we have illustrated a novel reactive navigation system based on shared intelligence approach and reinforcement learning. We have considered as input the information about the goal and distances from the obstacles computed through a clustering algorithm to discretize the space around the robot and used for representing the current state of the RL model. Therefore, we have successfully trained and tested, in a Gazebo simulation, different models, in particular, we found that the one exploiting the batch normalization and with a higher weight for the repulsive potential field. We have also considered several particular conditions for understanding if the models are able to generalize well. In addition to this, we have evaluated the models with dynamic obstacles that make the testing environment more challenging. In the last stage, we compare the overall system considering the current shared-intelligence system and one extended version combining reinforcement learning based on the four selected models according to the testing phase with the available policies. Furthermore, the preliminary results suggest that the proposed approach can enhance teleoperation according to the typical navigation metrics.
Future work aims to extend the current approach by implementing a system with memory-based functionality to provide long-term navigation (over extended periods). It is also possible to learn the model to deal with situations where the local cost map is subjected to noise to make the system more robust and avoid possible collisions in real scenarios.

References

[1] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[2] Gloria Beraldo et al. "Shared Intelligence for Robot Teleoperation via BMI". In: *IEEE Transactions on Human-Machine Systems* 52.3 (2022), pp. 400–409. DOI: 10.1109/THMS.2021.3137035.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[4] M. Morales. *Grokking Deep Reinforcement Learning*. Manning Publications, 2020. ISBN: 9781617295454. URL: https://books.google.it/books?id=IpHJzAEACAAJ.

[5] Bing Lu et al. "Hybrid Path Planning Combining Potential Field with Sigmoid Curve for Autonomous Driving". In: *Sensors* 20.24 (2020). ISSN: 1424-8220. DOI: 10.3390/s20247197. URL: https://www.mdpi.com/1424-8220/20/24/7197.

[6] Luíza Caetano Garaffa et al. "Reinforcement Learning for Mobile Robotics Exploration: A Survey". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021), pp. 1–15. DOI: 10.1109/TNNLS.2021.3124466.

[7] G. A. Cardona et al. "Autonomous Navigation for Exploration of Unknown Environments and Collision Avoidance in Mobile Robots Using Reinforcement Learning". In: *2019 SoutheastCon*. 2019, pp. 1–7. DOI: 10.1109/SoutheastCon42311.2019.9020521.

[8] Shirel Josef and Amir Degani. "Deep Reinforcement Learning for Safe Local Planning of a Ground Vehicle in Unknown Rough Terrain". In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6748–6755. DOI: 10.1109/LRA.2020.3011912.

[9] Céline Craye, David Filliat, and Jean-François Goudou. "RL-IAC: An Exploration Policy for Online Saliency Learning on an Autonomous Mobile Robot". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea: IEEE Press, 2016, pp. 4877–4884. DOI: 10.1109/IROS.2016.7759716. URL: https://doi.org/10.1109/IROS.2016.7759716.

[10] Haoran Li, Qichao Zhang, and Dongbin Zhao. "Deep Reinforcement Learning-Based Automatic Exploration for Navigation in Unknown Environment". In: *IEEE Transactions on Neural Networks and Learning Systems* 31.6 (2020), pp. 2064–2076. DOI: 10.1109/TNNLS.2019.2927869.

[11] Farzad Niroui et al. "Deep Reinforcement Learning Robot for Search and Rescue Applications: Exploration in Unknown Cluttered Environments". In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 610–617. DOI: 10.1109/LRA.2019.2891991.

[12] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[13] João Fabro et al. "ROS Navigation: Concepts and Tutorial". In: vol. 625. Feb. 2016, pp. 121–160. ISBN: 978-3-319-26052-5. DOI: 10.1007/978-3-319-26054-9_6.

[14] pal-robotics. *TIAGo, The robot that fits and adapts to your research, not the other way around*. URL: https://pal-robotics.com/robots/tiago/.

[15] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. O'Reilly Media, Inc., 2019. ISBN: 1492032646.

[16] Claudio Scheer, Larissa Guder, and Dalvan Griebler. "Deep Learning in Agriculture: A Systematic Literature Review". PhD thesis. Aug. 2019. DOI: 10.13140/RG.2.2.17367.19363.

[17] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[18] "The Publications and Writings of John W. Tukey". In: *The Annals of Statistics* 30.6 (2002), pp. 1666–1680. ISSN: 00905364. URL: http://www.jstor.org/stable/1558736 (visited on 06/12/2022).

[19] Brian S. Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis*. 4th. Wiley Publishing, 2009. ISBN: 0340761199.

[20] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: An Efficient Data Clustering Method for Very Large Databases". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 103–114. ISBN: 0897917944. DOI: 10.1145/233269.233324. URL: https://doi.org/10.1145/233269.233324.

[21] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[22] K. Fukunaga and L. Hostetler. "The estimation of the gradient of a density function, with applications in pattern recognition". In: *IEEE Transactions on Information Theory* 21.1 (1975), pp. 32–40. DOI: 10.1109/TIT.1975.1055330.

[23] Carlos Sampedro et al. "Laser-Based Reactive Navigation for Multirotor Aerial Robots using Deep Reinforcement Learning". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1024–1031. DOI: 10.1109/IROS.2018.8593706.

[24] Hagen Kleinert. *Path Integrals in Quantum Mechanics, Statistics, Polymer Physics, and Financial Markets*. 3rd. WORLD SCIENTIFIC, 2004. DOI: 10.1142/5057. eprint: https://www.worldscientific.com/doi/pdf/10.1142/5057. URL: https://www.worldscientific.com/doi/abs/10.1142/5057.

[25] Herbert Edelsbrunner. "Alpha Shapes a Survey". In: 2009.