POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Autonomous Robot Exploration using Deep Learning: An Experimental Analysis

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Marco Premi**

Student ID: 941388
Advisor: Prof. Francesco Amigoni
Co-advisors: Dr. Matteo Luperto
Academic Year: 2021-2022

Abstract

Among the different research fields in robotics, autonomous mobile robotics has been actively addressed in the last years.
Autonomous exploration is one of the most important tasks that an autonomous robot, deployed in an initially unknown environment, must accomplish. The robot, with no prior information about the environment, has to choose where to move and consequently the best strategy to explore the environment in order to build its map incrementally. Over the years, different strategies have been proposed and developed. Even if classical techniques proved to be mostly successful, a recent research thrust aims to develop Machine Learning and in particular Deep Learning techniques to address the exploration problem, because of the performance achieved by these approaches in other fields.
The purpose of this thesis is to compare classical and Deep Learning algorithms for exploration in order to understand what are the positive and negative sides of the different techniques. We compare them on different tasks and environments, obtaining comparable results for the classical and learning algorithms in most of the metrics considered. Results also highlight the difficulties faced by some learning algorithms when tested in more complex environments than those used in training.

Sommario

La robotica mobile autonoma è stata molto studiata negli ultimi anni.
Uno dei compiti più importanti che un robot autonomo, posto in un ambiente inizialmente sconosciuto, deve essere in grado di compiere è l'esplorazione. Il robot, senza alcuna conoscenza pregressa dell'ambiente, deve scegliere dove muoversi e conseguentemente la miglior strategia per esplorare l'ambiente stesso in modo da costruirne incrementalmente la mappa. Lungo gli anni diverse strategie sono state proposte e sviluppate. Anche se le tecniche classiche hanno dimostrato di avere successo nella maggior parte dei casi, una recente tendenza di ricerca mira a sviluppare algoritmi di Machine Learning e in particolare di Deep Learning per risolvere il problema dell'esplorazione, in virtù delle prestazioni raggiunte da queste tecniche in altri campi.
Lo scopo di questa tesi è quello di confrontare algoritmi di esplorazione classici e algoritmi Deep Learning in modo da capire quali sono gli aspetti positivi e negativi delle differenti tecniche. Nella tesi abbiamo comparato questi algoritmi in base a compiti diversi e in ambienti diversi, ottenendo risultati simili per gli algoritmi classici e per quelli che fanno uso di learning per la maggior parte delle metriche considerate. I risultati ottenuti evidenziano anche le difficoltà incontrate da alcuni algoritmi che usano tecniche di learning quando vengono testati in ambienti più complessi rispetto a quelli di training.

Ringraziamenti

In questo spazio vorrei dedicare un pensiero a tutte le persone che hanno contribuito, in maniere diverse ma fondamentali, alla realizzazione di questa tesi.
In primis, un ringraziamento speciale va al mio relatore, il Prof. Amigoni Francesco per la disponibilità e l'attenzione che mi ha dedicato in questo percorso.
Grazie anche al mio correlatore, il Dr. Luperto Matteo per i preziosi consigli e suggerimenti su come modificare la tesi per renderla migliore.
Ringrazio mia mamma e mio padre, per il supporto durante questi anni di università, senza di loro sarebbe stato impossibile seguire questo percorso.
Un grazie con tutto il mio cuore, il grazie sicuramente più grande, alla mia ragazza Martina, che mi è sempre stata vicina nei momenti più difficili e tesi degli ultimi anni; probabilmente non ce l'avrei mai fatta senza il tuo aiuto.
Ringrazio anche tutti gli " Idiosincratici ": Gianty, Giordie, Guaro, Gwen, Chiara, Louise, Eleonora e Nick. I momenti di leggerezza, ma anche di serietà che abbiamo condiviso assieme ultimamente sono stati davvero fondamentali per arrivare a questo traguardo. Tra di loro un ringraziamento particolare a Gianty, per l'amicizia che ha saputo superare mille difficoltà.
Grazie anche a Marco, Sarah, Stefano e Margherita per il tempo trascorso insieme, ricco di spensieratezza e risate.
Un ringraziamento a Martina B. soprattutto per l'amicizia ritrovata.
Infine, grazie a Martina, Dylan e Camillo per i bellissimi momenti passati assieme e per quelli che verranno.

Contents

1 | Introduction

Among the different research fields in robotics, autonomous mobile robotics has been actively addressed in the last years. A robot, in order to be called autonomous, must be able to move in an environment or to fulfill a task without the need for continous human intervention. One of the most important tasks that a robot has to perform is *exploration* of initially unknown environments. In this task, the robot is required to autonomously visit a possibly unknown environment and incrementally build a map of it, with the data collected during the exploration.

Different techniques have been developed in the last years in order to make a robot autonomously explore an unknown environment. Even if these classical techniques proved to be successful in many cases, a recent research thrust aims to develop Machine Learning and in particular Deep Learning algorithms to address the exploration problem, because of the performance achieved by these methods in other fields. Deep Learning techniques are used for example with successful results in speech [91] or image recognitions [94].

Even if Deep Learning is proposed in place of the classical paradigms, to date, these techniques have not been extensively compared. Deep Learning methods require a lot of data and in most cases when we compare these kinds of methods to classical ones they lack explainability [17]. After training on a lot of episodes (in order of hundreds of thousands), the cause of problems during exploration (like a collision, or a point-goal not reached), may not be found so easily. In addition, most of the Deep Learning exploration algorithms are studied only in simulation environments and only sometimes have also been tested in real-world environments. Most of the real-world experiments are, however, done in very simple environments [73].

The purpose of this thesis is so to compare classical exploration algorithms and algorithms that exploit the possibility offered by Deep Learning. Each of the two approaches offers downsides and upsides and because of the high computational power required to train a Deep Learning algorithm, it may be useful to understand when it is worth being used.

More precisely, in this thesis, we survey different works that address the exploration problem, classifying them into different categories in order to understand how Deep Learning and classical techniques can be compared.

When we talk about exploration, Deep Learning can be used in different parts of the exploration framework. Traditionally, during exploration, maps are built with *SLAM (Simultaneous Localization and Mapping)* [112] algorithms. With SLAM we refer to the task of building a map while trying to localize the agent in it. Maps built with this method can be enhanced in order to simplify exploration. Predictions of unseen areas can be added to the maps with numerical or Deep Learning techniques. In [110] the authors use a method to infer the unobserved portion of a map from the observed portion. Deep Learning techniques can be used to make such predictions, like in [100] or [74]. In these two works, the authors use a Deep Learning network that takes as input the current observed map and outputs predictions of the unknown parts of the environment.
Works like [48] propose Deep Learning algorithms that, working similarly to SLAM, produce as output the map update with the current observations (and no predictions) and the pose estimate.
During exploration of an unknown environment, selection of the next location to explore is a very important step. Most of the classical methods proposed to select the next exploration location are based on the concept of *frontiers*, defined as the regions between free and unexplored space. The most famous algorithm is called frontier exploration [121] and has been proposed in 1997. With this approach, the robot moves to the closest frontier, until no more frontiers are reachable. Beyond this initial strategy, the next frontier to be visited can be selected in multiple ways, i.e., randomly or by selecting the frontier point that maximizes the amount of unknown area that can be seen from it [43]. Selection of the next exploration location, as already said, can be done in a classical way with a frontier exploration strategy or with Deep Learning algorithms. An example of a Deep Learning algorithm trained to learn what is the next location to explore is reported in [48]. Authors train a *Deep Reinforcement Learning* policy that rewards increase in area coverage. The policy takes as input the current map and pose in order to output the next point goal to reach. In the literature, other works follow this idea, rewarding other elements like giving positive rewards if actions are smooth or if the current observations add obstacles of free-spaces to the map (e.g., [55] and [50]).
Classical frontier exploration techniques can also be enhanced with Deep Learning. For example authors of [66] use a Deep Learning network to learn a policy that selects the best frontier point, instead of simply selecting the nearest frontier point.
Once the next exploration point has been selected, different techniques can be deployed in order to reach it. Agents can for example use classical path planning algorithms like A* or Dijkstra in order to plan a path from their current position to the point to be explored (e.g., [111] and [106]). Deep Learning techniques can be used also in this case. For example authors of [60] use a Deep Learning network that takes as input the point

selected for exploration and a description of the environment in order to output the action that must be executed by the agent in order to reach that point.
One key element to keep in mind when developing Deep Learning algorithms is their tendency to exploit imperfections [17]. Because these exploration algorithms are trained in a simulator and not in the real world, they may learn how to exploit imperfections of the simulator that can't be found in the real world. If simulation environments where Deep Learning algorithms are trained are too simple, the agent may not perform well when deployed in more complex environments. In other cases, agents may learn to move in ways that can't be replicated in reality. In [16], for example, agents in Habitat simulator [84] learn to slide along obstacles when colliding, exhibiting a behavior that can't be replicated in the real world. Deep Learning algorithms in many cases require high computational power in order to be trained, for example in [6] where authors reported a training with 72 parallel threads on a system with 8 Nvidia V100 GPUs that lasted 24 hours. While classical algorithms can be simply deployed in the exploration environments without particular computation needs, Deep Learning algorithms require, as just said, a long training and may suffer from generalization issues in new environments.
Because of the reasons just stated, the purpose of this thesis is to compare Deep Learning and classical exploration algorithms, in order to understand what can be the difficulties faced by these two kinds of approaches. They are compared on different tasks in order to understand if all the exploration steps can have big performance improvements with the introduction of Deep Learning algorithms, or if these new techniques have a material impact only when used for specific jobs (like map prediction). Furthermore, some of the exploration steps, like the navigation step, are already solved with the optimal solution with classical algorithms like A*. In this case, we must understand if Deep Learning algorithms are worth using. Results of the classical and learning algorithms obtained during the comparison are comparable in most of the metrics considered. Results also highlight the difficulties faced by some learning algorithms when tested in more complex environments than those used in training.

In order to fulfill the purpose previously highlighted, the thesis is structured as described in the following lines.
In Chapter 2, we describe the state of the art regarding the exploration strategies that can be found in the literature. The focus of this section lies in the identification of classical and Deep Learning techniques that can be used in different modules of the exploration framework.
In Chapter 3, we describe the problem analyzed in the thesis work, starting from a de-

scription of how classical and Deep Learning techniques can be used in order to solve the exploration problem. Then we discuss the purpose of the thesis, which is the comparison of classical and Deep Learning algorithms in order to understand what are the upsides and downsides of every implementation. Furthermore, we describe the algorithms chosen for the comparison and how the comparison is done.
In Chapter 4, we describe the main software and simulators used in order to compare the algorithms and how they have been modified to suit our needs.
In Chapter 5, we describe the tests done in order to compare the algorithms and the results obtained.
In Chapter 6, we briefly summarize the purpose of the thesis, highlighting the results obtained. We finally propose other elements to be taken into consideration for future developments.

2 | State of the art

Autonomous mobile robotics is a research field that has received significant attention in the last years [107]. An autonomous mobile robot is an *agent* that, through different *sensors*, perceives the *environment*. A robot, in order to be *autonomous*, must be able to make movements and fulfill tasks like "pick an object" or "move to point A" without any human intervention. *Actions* are made through *actuators* (like electric motors). The task of *exploration* is, of course, one of the most important tasks that an autonomous agent has to solve. In order to solve the *exploration for map building problem*, the robot has to visit a possibly unknown environment (in some cases the agent has access to partial information about it) and incrementally build a *map*. The maps (in most of the cases 2D maps) represent all the free areas and the obstacles present in the environment. There exist different map types associated to each environment (Section 2.2) and are built using the information collected by agent's sensors when it is moving in the environment. During the exploration, the agent, has also to localize itself and the inputs received from the sensors and the partial map built till that moment, moreover, the robot must be able to decide where to go next without any human supervision.

To solve this problem some *exploration strategies* have been proposed during past years and most of the algorithms works even if no knowledge of the environment is given to the agent before the exploration process.

One of the most popular algorithm for this task is called frontier exploration [121], which makes the agent move to the closest *frontier* (defined as a region between free space and unexplored space). In other methods like [64] the agent moves to "good" positions where "good" is referred to the quality and quantity of information that will be available when arrived at the specific location. In this method, at each iteration the algorithm first generates a set of potential candidates and next it evaluates them according to the expected information gain that will be sensed in the candidate position and the cost required to navigate to the new position.

In the last few years, Deep Learning techniques have been proposed in order to solve the exploration problem. Deep Learning algorithms have recently achieved important success among different fields like speech recognitions [91], image [94], or video recognition [120].

Among these, Deep Reinforcement Learning algorithms have been proposed to train agents that can play complex games like Go [108], Atari [85], Starcraft [72], ...
Deep Learning networks has shown to be effective in learning and integrating information from different inputs (like cameras, laser, GPS, ...); consequently, researchers tried to use these techniques also with mobile robots. Authors of preliminary works like [56] provide a Deep Learning algorithm able to recognize doors while the agent is moving in the environment and to create a roadmap based on the recognized doors. Authors of [44] proposed a Reinforcement Learning algorithm able to learn a policy to navigate from an initial to a target position by generating a collision-free path. However, different types of approaches have been proposed to solve the problem of exploration using also Deep Learning methods. Figure 2.1 shows an overview of the steps used for solving this tasks and the most popular approaches used for each one of these steps.

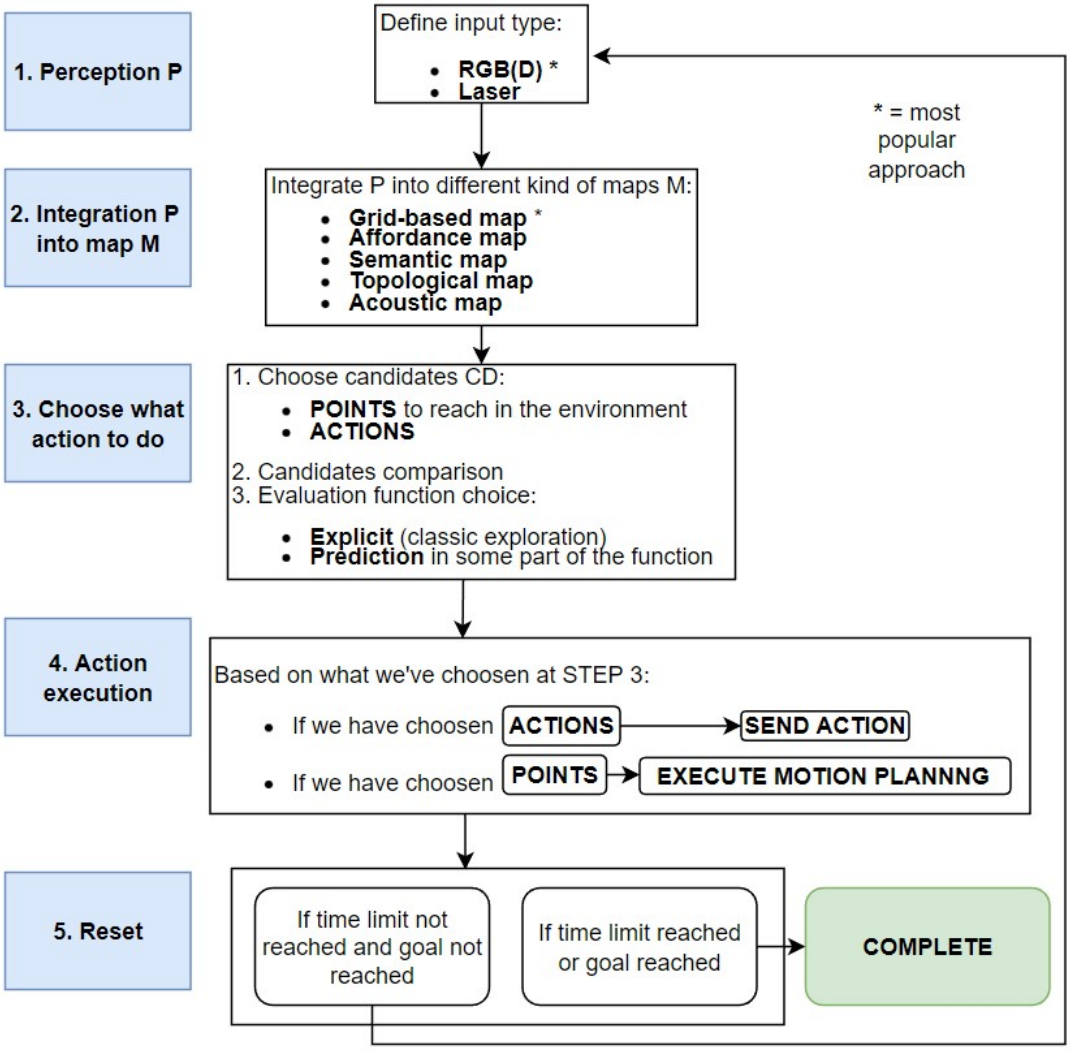


Figure 2.1: Steps of the exploration process.

The exploration steps described in Figure 2.1 are:

1. **Perception P:** the agent observes the environment and collects the input data (RGB images, depth images, or laser scans).
2. **Integration P into map M:** the data collected at the previous step are used to build maps of different types. Some algorithms that don't require an explicit memorization of the map are also proposed.
3. **Choose what action to do:** the agent, given the data collected and the map built, has to select a list of candidates, i.e., the next thing to do. Candidates can be:

   - POINTS to reach in the environment.
   - ACTIONS to do (like move left, move right, stay in place, ...).

   The agent has to compare the elements in the list of candidates and can do this with an explicit evaluation function or with the help of predictions (for example it can predict unseen areas in the map and use this information to select the best candidate).
4. **Action execution:**

   - If at step 3 the agent has selected a POINT, it has to execute motion planning to reach that point.
   - If at step 3 the agent has selected ACTION, it has to send the action command to the actuators.
5. **Reset:**

   - If time limit is not reached and goal is not reached, go back to step 1.
   - If time limit or goal is reached, algorithm stops execution.

Deep Learning algorithms can be used in different parts of the formalization proposed. Some authors use them in order to add predictions to the maps (e.g., [97] and [100]) or to add class labels (e.g., [49], [50] and [51]) to the pixels in the map in order to identify specific objects to be reached. Others employ these techniques to select the next exploration point or action (e.g., [48]). And others employ Deep Learning to learn a navigation policy used to reach the point selected among the list of candidates (e.g., [60]). In the next sections, we provide a detailed description of these main points and of how they were addressed in the literature.

2.1. Sensors

In order to explore the environment in an autonomous way and to build a map, the robot must be able to perceive what is around itself and determine its current localization through sensors. The robot acquires different kinds of inputs that are used and processed in multiple ways by the different algorithms.

- **RGB sensor:** a sensor made of an RGB camera.
- **Depth sensor:** a sensor that measures the distance to the surrounding objects.
- **RGBD sensor:** a sensor made of a union of RGB cameras and depth (D) sensor. It is used to associate the image coming from the RGB sensor to a depth channel, in order to combine each pixel with the distance to the corresponding object.
- **Laser range finder:** a sensor that uses a laser beam to determine the distance to an object. They are also known as *lidars*.
- **GPS:** the sensor used to measure the position of the robot in the environments.
- **IMU - Inertial Measurements Sensor:** a sensor used to measure orientation and accelerations.
- **Bump sensor:** a sensor used to sense contact with obstacles.

2.2. Map representation

A robot that is autonomously exploring the environment is required to acquire and update a map representation of the surrounding environments, to perform all the operations and calculations needed to define the exploration path. Different representations of the maps have been proposed and commonly used; among so, relevant ones are (also shown in Figure 2.2):

- **Grid-based map:** the map is divided into cells of fixed dimensions. Each cell has a value that represents the probability of the cell to be occupied by an obstacle. Different thresholds can be set to this value in order to consider the cell occupied or not. Another example of a grid-based map can be the acoustic map, a grid where each cell contains a value that represents audio intensity in that location.
- **Topological map:** a graph is used to represent the map. Even if commonly used, there is not a standard definition of what nodes and edges represent, and their meaning varies based on the kind of applications developed. For example in [98] nodes correspond to locations and edges represent the existence of a path between

two edges.

- **Semantic map:** the map is divided into cells of fixed dimensions. Each cell has a value that represents if the cell is an obstacle, explored or contains an object of the corresponding category (from a list of possible predefined categories). Another example of a semantic map can be the affordance map, a grid where each cell contains a value that represents the probability of that location to allow a specific, given, affordance. Examples of affordances are: "pickupable", "sliceable", and so on.

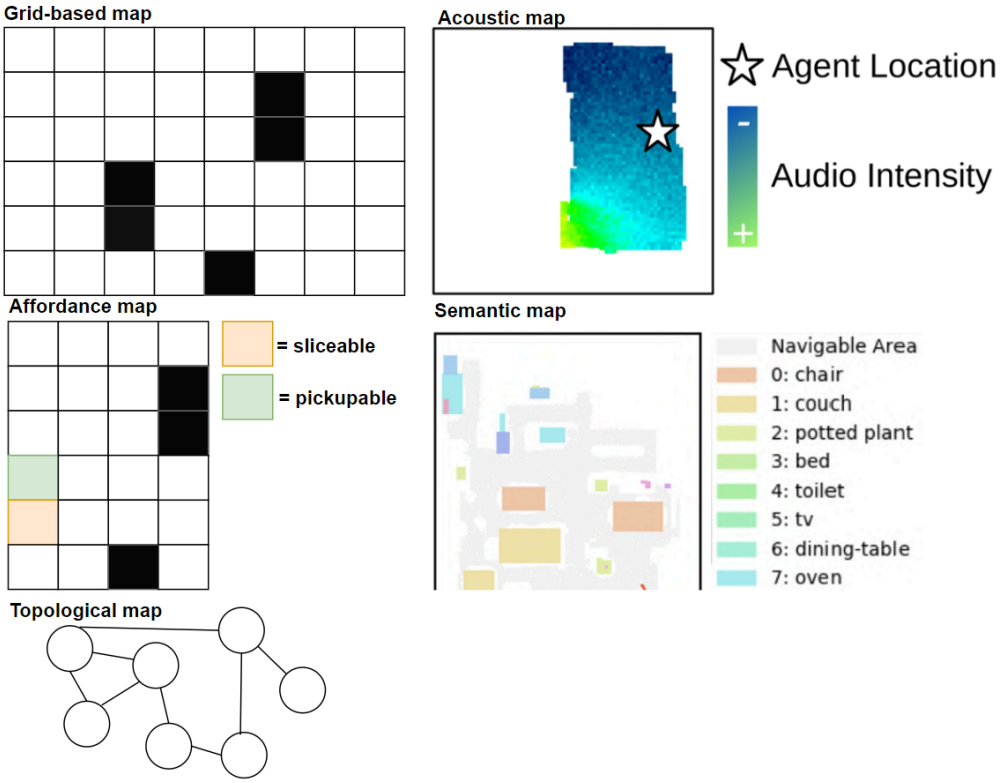


Figure 2.2: Examples of the different map representation: grid-based map and acoustic map are taken from [52], affordance map and semantic map are taken from [49].

Other contexts do not use a map and are mapless. In this case, actions to take are derived immediately after sensor readings.

2.3. Exploration tasks

The most common exploration tasks are:

- **Exploration:** the standard exploration for map building task, where the agent is required to explore the environment and build a map as accurate as possible (e.g., [106], [48], and [97]).
- **Point-goal driven exploration**: the agent is required to explore the environment and build a map used to reach the point in space provided as input (e.g., [60], [48], and [97]).
- **Object-goal driven exploration**: the agent is required to explore the environment and build a map used to reach a specific object (e.g., refrigerator, car, keys, ...) provided as input (e.g., [49], [122], and [78]).
- **Area-goal driven exploration**: the agent is required to explore the environment and build a map used to reach a specific area in the environment (e.g., kitchen, garage, foyer, ...) provided as input (e.g., [90]).
- **Audio-goal driven exploration**: the agent is required to explore the environment and build a map used to reach the source of the emitted sound (e.g., [52]).
- **Image-goal driven exploration**: the agent is required to explore the environment and build a map used to reach a specific image provided as input (e.g., [55], and [101]).
- **Interaction-goal driven exploration**: the agent is required to explore the environment and build a map used to make it possible for the agent to interact with as many objects as possible (e.g., [51]).
- **Objects detection exploration**: the agent is required to explore the environment and build a map used to make it possible for the agent to correctly label as many objects as possible (e.g., [51]).

In most cases an algorithm designed to accomplish a specific exploration task is unable to achieve also another task. Some exceptions are possible, for example [48] and [97] with little changes can be used for both exploration and point-goal driven exploration.
In [55] the authors explicitly want to create an algorithm that is not only able to solve the standard exploration task but it is also able to solve the other exploration tasks. In the proposed algorithm the agent first explores the environment in order to solve the exploration for map building task end then, once exploration is finished, the agent is given a different task, for example object-goal driven exploration.

2.4. Exploration framework modules

From the literature, in order to understand how exploration tasks can be solved, we have identified three macro modules useful to classify the different modules required to perform exploration, as shown in Figure 2.3. These modules are obtained from the exploration steps already formalized in Figure 2.1.

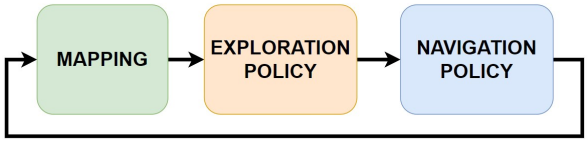


Figure 2.3: High-level schema of the exploration framework.

Each one of these blocks could be implemented in several ways. Each framework for performing exploration should select an algorithm or a methodology (or more than one) for each block. Most of the proposed frameworks start by generating a map from the current observations (*mapping* module). Then they use an *exploration policy* to select the next point goal that must be reached in order to solve one of the proposed exploration tasks. In order to reach this point in space they use a *navigation policy*.
Mapping, exploration policy, or navigation policy modules can be implemented with Deep Learning networks or use classic methods. Some of the algorithms, for example, use Deep Learning methods only in one of the modules, preferring to use well-established classic techniques in the other modules.
Other exploration frameworks instead consider exploration policy and navigation policy as a single module, that can be trained *end-to-end* taking advantage of Deep Learning neural networks, as shown in Figure 2.4.

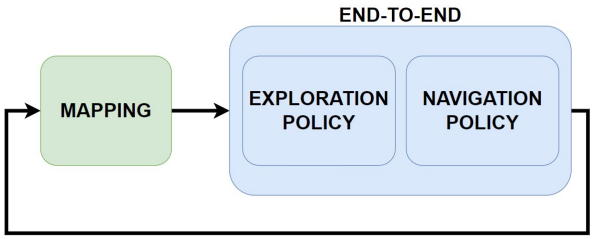


Figure 2.4: High-level schema of the exploration framework with end-to-end module.

Different types of algorithms are used to implement the mapping module, often fusing this step with other ones: some of them simply update the map with the current observations, others try to predict the unseen parts of the maps, still others add semantics information over it.

Paper	Mapping	NAVIGATION POLICY	EXPLORATION POLICY	END-TO-END	NOISE
[66]	Deep Learning	Classical	Classical	X	NO
[46]	Deep Learning	Classical	Classical	X	NO
[100]	Deep Learning	Classical	Classical	X	NO
[106]	Deep Learning	Classical	Classical	X	NO
[48]	Deep Learning	Deep Learning	Deep Learning	X	YES
[74]	Deep Learning	X	X	X	NO
[92]	Classical	Classical	Deep Learning	Deep Learning	NO
[90]	Deep Learning	Deep Learning	Deep Learning	X	NO
[75]	Deep Learning	Classical	Classical	X	NO
[39]	Deep Learning	Classical	X	X	NO
[97]	Deep Learning	Deep Learning	Deep Learning	X	YES
[110]	Classical	Classical	Classical	X	NO
[52]	Classical	Classical	Deep Learning	X	YES
[82]	Classical	Classical	Classical	X	NO
[124]	Deep Learning	Deep Learning	Classical	X	NO
[55]	Classical	X	X	Deep Learning	YES
[101]	Classical	Deep Learning	Deep Learning	X	NO
[41]	Classical	X	X	Deep Learning	NO
[60]	Classical	Deep Learning	Classical	X	YES
[76]	Classical	X	X	Deep Learning	NO
[49]	Deep Learning	Classical	Deep Learning	X	YES
[50]	Deep Learning	X	X	Deep Learning	NO
[89]	Deep Learning	X	X	Deep Learning	YES
[122]	Mapless	X	X	Deep Learning	NO
[111]	Classical	Classical	Deep Learning	X	NO
[96]	Deep Learning	Classical	Classical	X	NO
[78]	Deep Learning	Classical	Classical	X	NO
[71]	Mapless	X	X	Deep Learning	NO
[51]	Deep Learning	Classical	Deep Learning	X	NO
[123]	Classical	Deep Learning	Classical	X	NO

Table 2.1: Some representative papers from the literature.

Table 2.1 classifies a representative sample of papers from the literature in order to build a framework for exploration. With *classical* we refer to the kind of algorithms that don't use Deep Learning. In mapping, navigation policy, and exploration policy columns we have shown if the authors of the paper have used Deep Learning algorithms or classical ones in the module. Mapping module is covered in detail in Section 2.4.2, exploration policy module in Section 2.4.3, and navigation policy module in Section 2.4.4. In the papers that use the end-to-end Deep Learning module (considered in Section 2.4.5), as already said, exploration policy and navigation policy are considered as a single module, and because of this are marked with an "X". In the noise column we have shown if the authors of the paper have considered noise in the simulation settings, as described in Section 2.5.1.
Looking at Table 2.1 we can see that most of the authors prefer to consider exploration and navigation policies as separate modules, given that end-to-end module is used in only 8 cases. In more than half of the examined papers (17 cases) Deep Learning algorithms are used to implement the mapping module, while Deep Learning is less used in navigation policy module (7 cases) and exploration policy module (9 cases.)

2.4.1. Deep Learning algorithms

Different Deep Learning algorithms can be used in all the modules of the framework; some for example can be used for map prediction, others for training a navigation or exploration policy.
The Deep Learning algorithms used to train the policies described in the papers belong to these families:

- **Reinforcement Learning**: used in exploration policy, navigation policy, end-to-end modules.
- **Imitation Learning**: used in navigation policy and end-to-end modules.
- **Supervised Learning**: used in mapping, exploration policy, end-to-end modules.
- **Unsupervised Learning**: used in mapping module.
- **Self-Supervised Learning**: used in mapping, navigation policy modules.

We now briefly describe these frameworks.

Reinforcement Learning (RL)

The typical structure of a Reinforcement Learning algorithm (RL) is composed of two main components: *agent* and *environment*, as shown in Figure 2.5.

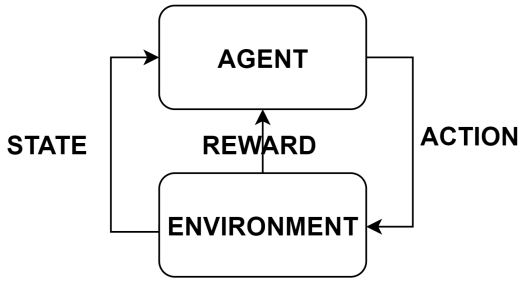


Figure 2.5: High-level schema of Reinforcement Learning.

The goal of the method is to use the observations collected from the environment (the inputs of the Deep Learning environment) to maximize the reward function.
The agent first receives a *state* from the environment. In the exploration framework here analyzed, this state is composed by either by RGB images (e.g., [97]), depth images (e.g., [97]), laser readings (e.g., [60]) or entire maps (e.g., [50]). The agent, based on the amount of experience accumulated till that moment, takes an action (move forward, rotate, ...) in the environment. After that, the agent receives from the environment the next state and the reward obtained by acting in that way. The agent so trains the *policy* in order to maximize the reward obtained.
In RL the agent continuously learns from experience in the environment until it has reached the goal or the maximum number of steps allowed.
In the examined papers ([48], [49], [50], [51], [52], [60], [63], [76], [90], [97], [101], [111], [122], and [123]) different reward functions recognize as important different aspects in the proposed exploration tasks (Section 2.3), most of them use:

- **Positive reward:** if area seen increases; if reduction in distance to goal (local or global); if current observation adds obstacles or free-spaces to the map; if accuracy in map prediction increases; if more than a certain percentage of the area has been explored; if agent gains confidence in predicting object categories; if reduction in temporally inconsistent predictions (objects in the scene are labeled different in subsequent frames); if goal is reached fast; if actions are smooth (no excessive rotational velocity).
- **Negative reward:** with a collision/invalid states; per timestep (small, but used to avoid unnecessary movements).

Most of the Reinforcement Learning-based frameworks proposed in the thesis are trained

with Proximal Policy Optimization (PPO) [103]. Some of them also take advantage of Decentralized Distribute Proximal Policy Optimization (DD-PPO) [118]. DD-PPO is a distributed Reinforcement Learning method that allows training in resource-intensive simulated environments on multiple machines. The possibility of training simultaneously on different environments allows the agent to obtain more experience more quickly, but uses a lot of computational resources.
Writing a reward function may be tricky and sometimes a conceptually good function may lead to unwanted behaviors. For example if we reward an exploring agent only for seeking novelty, we can have a situation as the one described in [45]: if we place a television continuously reproducing new images the agent is trapped by its reward function and keeps observing it without exploring the environment anymore.
Reinforcement Learning algorithms are used in the modules that consider agent's movements in the environments: exploration policy, navigation policy, and end-to-end. They are never used in the mapping module.

Imitation Learning

As just described, RL algorithms require pre-specified reward functions. Most of the time specifying a reward function is a difficult task and it can be convenient to instead use Imitation Learning (IL) techniques. With Imitation Learning, agents learn a policy from experts that provide the agent a set of demonstrations. The agent, once having accessed this set of demonstrations, tries to learn a policy by following what experts do. There are some situations where it is very useful to use Imitation Learning instead of other techniques, for example when, for an expert, it is easier to demonstrate the desired behavior rather than writing an explicit reward function. For a human, for example, the act of navigation from one point to another point in space is a simple and instinctive behavior, that can be in most situations very difficult to formalize.
In this case, the environment contains a set of *states*, a set of *actions*, a *probability transition model*, and an unknown *reward function*. The agent performs actions in the environment, trying to optimize its policy, having access to the expert's optimal policy.
Different implementations of Imitation Learning-based framework exist like Direct Policy Learning [58] or Inverse Reinforced Learning [35], but the one used in the examined papers is Behavioral Cloning [116], used in [48] and [55].
Imitation Learning (and Behavioral Cloning in particular, as just said) is used in the navigation policy and end-to-end modules. The algorithm is strongly based on the concept of "experts showing how to solve a task" , and because of this it can be a good choice for learning how to move in an environment, but it can't be used in mapping module.

Behavioral Cloning represents a way in which Imitation Learning can be implemented. A high-level schema of the algorithm is presented in Figure 2.6.

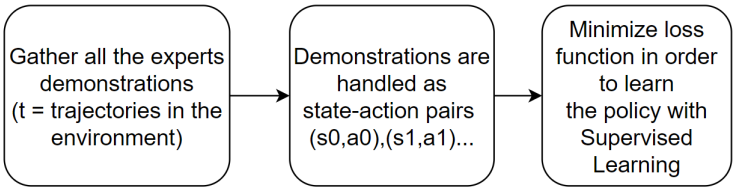


Figure 2.6: High-level schema of Behavioral Cloning algorithm.

The agent tries to learn a policy as much similar as possible to the one provided by experts in the environments. This method is not robust to training errors and often fails in settings where not enough experts demonstrations are provided. One example of this behavior is when the agent visits a state never visited by the experts (and so not considered in training) and so it has no data on how to behave from that state. There is a high correlation between the amount of available data and the quality of the policy obtained with Behavioral Cloning. As just described Behavioral Cloning is simple to implement, but its issues must be carefully considered.

Supervised Learning

Supervised Learning is a class of algorithms where the training selects the function that best describes the input data. Input data during training are provided with labels that describe them. The algorithm learns a function that must be able to associate labels to unlabeled data. There exist two main ways of implementing Supervised Learning:

- **Classification:** algorithms used to classify discrete values (like true or false, dog or cat, ...).
- **Regression:** algorithms used to estimate continuous values (like velocities, meters, ...).

Supervised Learning paradigm can be used to learn a policy for exploration. The network used to train the policy, for example, can take as input a map and provide as output the optimal action from the current point in the map. The training dataset in this case requires images of the map with the current agent location labeled with the optimal action, as shown in Figure 2.7.

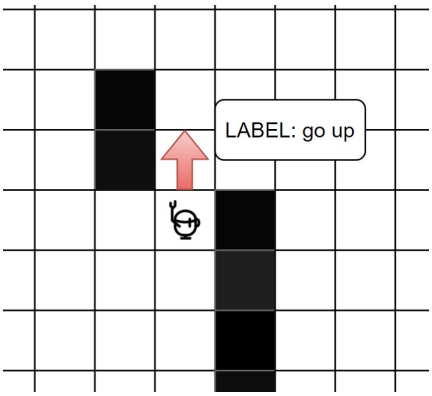


Figure 2.7: An example of the use of Supervised Learning for exploration.

Supervised Learning algorithms are used in mapping (e.g., [90]), exploration policy (e.g., [90]) and end-to-end (e.g., [41]) modules.

Unsupervised Learning

In Unsupervised Learning the situation is the opposite with respect to what we have in Supervised Learning. In this learning task, the input data have no labels or classification associated. The goal is to find some underlying pattern that can characterize the dataset. One task where Unsupervised Learning is usually included is segmentation. The input is not associated with any labels, and so the best way to find a pattern is to find a function able to cluster the input.

Unsupervised Learning is used in mapping module in generative networks like in Section 2.4.2 where the proposed networks want to learn a function able to generate the unseen parts of the maps (used in works of [39], [74], and [106]). Generative networks are a very common choice in papers that want to predict unseen parts of the maps with Deep Learning techniques.

Self-Supervised Learning

Self-Supervised Learning is a class of algorithms similar to Unsupervised Learning, but that tries to solve problems traditionally targeted by Supervised Learning (e.g., classification). Also in this class of problems the algorithm has to solve classification or regression problems as previously described. In this case, labels are not directly provided during training stage. The algorithm has to extract the labels from the samples in order to be able to solve the problem.

Self-Supervised Learning algorithms are used in mapping ([50] and [96]) and navigation policy modules ([101]).

2.4.2. Mapping

Most of the algorithms that we have investigated use an explicit map to represent the environment. Maps are created or updated at the beginning of each algorithm step in the environment, after having received inputs of different types (RGB images, depth images, current position, estimate of the current position, actions...). In the simplest case maps are simply updated adding the input received to the current map without adding prediction or other kinds of information. Map predictions can be obtained with numerical techniques or with Deep Learning algorithms.
Before the arrival of Deep Learning, mapping was traditionally obtained with *SLAM (Simultaneous Localization and Mapping)* [112]. With SLAM we indicate the task of building the map of the environment while trying to localize the agent in it.
Even algorithms that use Deep Learning techniques for map prediction or map labeling sometimes take maps generated with SLAM and then add predictions or other information to them, as we will see later in this section.
SLAM can be formally described taking into consideration:

- Path (sequence of locations): $X_T = x_0, \ldots, x_T$
- Odometry sequence: $U_T = u_0, \ldots, u_T$
- Sensors measurement sequence: $Z_T = z_0, \ldots, z_T$

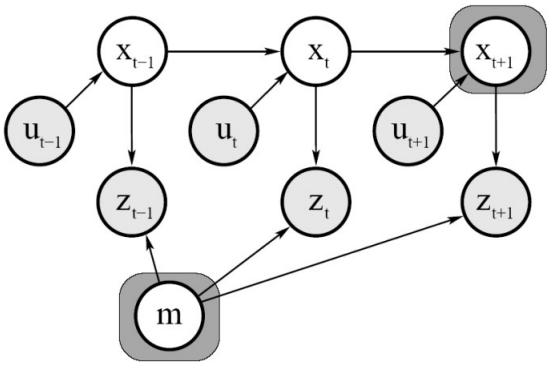


Figure 2.8: Graphical representation of SLAM, from [30].

In Figure 2.8 we can see a graphical representation of the SLAM problem where the arcs represent the causal relationships and shaded nodes what can directly seen by the robot. The SLAM problem is so formalized as the problem of recovering a model of the world (the map m) and the sequence of robot locations (X_T) from what it can directly observe (odometry and sensors data).
Authors from [92], for example, use SLAM with information coming from 3D sensor and

odometry sensor to update the map at each step. This map is then provided to the exploration policy.
In [52] authors use as inputs depth images and sound received from left and right microphones to build and update two maps (occupancy map and acoustic map) as the agents move in the unmapped environment.
Deep Learning networks can be used in mapping, even if in most of the cases they are used for map predictions (taking as input map created with SLAM).
One example of Deep Learning networks used for mapping is the one provided in [89], where authors use U-net [99] having as input only RGB images in order to output affordance maps.
Another example is the mapper structure presented in [48], similar to U-net. It takes as input an RGB image and produces as output an egocentric top-down 2D spatial map.
Deep Learning networks can be used in order to obtain semantic maps, as done in [49], [50], [51]. In these papers, authors use the same structure (shown in Figure 2.9), built with a pretrained Mask R-CNN network ([68]).

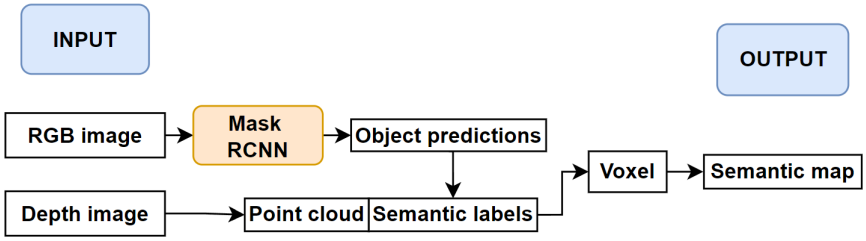


Figure 2.9: High-level schema of the semantic mapping module implemented in [49] and [50].

Mask R-CNN receives as input an RGB image, used to produce the object predictions as output. The output values combined with the values from depth images are used to produce the semantic map. It is interesting to note that with the schema presented in Figure 2.9 authors can use a pretrained network.
We now briefly describe how predictions can be added to the maps with or without Deep Learning techniques.

Map prediction without Deep Learning

Techniques that do not depend on Deep Learning can be used to add predicted elements to the maps.
In [110] authors use a method that takes as input the observed portions of the map (white) to infer the unobserved portions (red), as shown in Figure 2.10.

Figure 2.10: Example of map generated by [110]: in white the observed portions of the map, in red the unobserved portions inferred.

The inference method developed in this paper is made of two components:

- **Heuristic-based perimeter prediction:** the algorithm proposed estimates the exterior boundary of the map. Observed grid-based map cells are used to predict unobserved portions.
- **Structural inference:** it uses the predicted perimeter and the observed portions of the map to infer the unobserved internal map. The main assumption under structural inference is that most environments are similar to other structures that exist. So they use a collection of general maps to match structures with the observed cells in the map.

This algorithm of course requires a lot of entries in the environments library in order to provide as many examples as possible and this situation has an impact on computation time.
In [82] authors use a method presented in [80] and [81] to predict the shape of partially observed rooms (partial grid-based map acquired till that moment), but in principle any other method that provides a geometrical estimate of room shape could be used. The method takes as input the partial grid-based map and outputs the predicted grid-based map. The main idea is to identify the shape of the parts that are fully observed in the environment and to use this knowledge of the structures to predict what can be the shape of partially observed rooms.

Map prediction with Deep Learning

Multiple kind of Deep Learning networks are implemented in the papers in order to predict maps or elements in the maps. The Deep Learning methods can use different kinds of inputs (RGB images, depth images, previous maps (sometimes obtained with SLAM), actions, previous actions). Some of the Deep Learning networks are pretrained, others must be trained on the specific task.
The most common networks architectures used are: Convolutional Neural Network (CNN)

[36], U-net [99], GAN [61], VAE [32], CVAE [32], and Sequence to Sequence [113].
The first example of algorithm trained to predict elements of a map is the one proposed in [100]. Authors of this work use a U-net network which takes as input the currently explored environment map (converted into an image) and outputs a prediction of the unknown parts of the environment.
Also in [75] the authors use a U-net architecture taking as input the current grid-based map and producing as output the prediction the unseen area.
In [97], the network is the one represented in Figure 2.11.

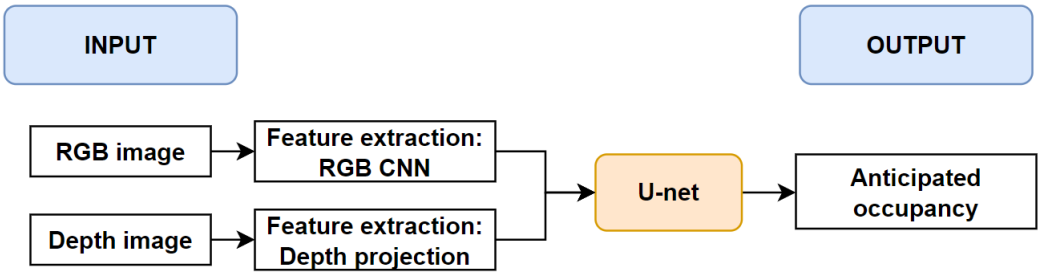


Figure 2.11: High-level schema of how map prediction is implemented in [97].

In this case, two inputs are provided to the network: RGB images and depth images, and the features extracted from them are combined for feature encoding. The output is the anticipated occupancy (tensor of probabilities which represent the probabilities of having obstacles in unseen areas).
Different *Generative Networks* (networks trained to generate new data very similar to the one contained in a specific dataset) like GAN, VAE and CVAE are used in several works. In [74] the authors train a GAN with Unsupervised Learning to perform map completion. The network takes as input the uncompleted map and outputs the complete map. In [106] the authors train a VAE with Unsupervised Learning in order to produce a prediction of a small part of the map. The network takes as input a four channel image representing the currently known map and the mask for the prediction region. It produces as output a single channel image representing the probability of obstacles. In [39], the authors use a CVAE network because of its capability of conditioning the generation process to different variables. Because of this, the network takes as input a sub-map and an action and produces as output a map conditioned on the action chosen.
Authors of [90] use a Sequence to Sequence network for map prediction. The network takes as input the current image (RGB), the previous semantic map and the previous actions in order to predict the current semantic map. The policy is trained with Supervised Learning.
Authors of [46], instead, use a CNN trained with Supervised Learning to predict the exit

locations of the floor plan. The training dataset of this algorithm is composed of labeled images where all the sub-images are labeled with 1 if they contain exit locations, 0 otherwise. The algorithm takes as input a blueprint image of the plan and outputs a smaller image where each pixel can have two values: 1 for exit locations, 0 otherwise (heatmap of estimated exit locations).

2.4.3. Exploration policy

During exploration in an unknown environment different tasks must be accomplished: the perception of environment information with agent's sensors, the integration of the inputs received into the current map and the selection of the most promising next goal location. The selection of next location to be reached by the robot is probably the most important component of the exploration strategy and it is done by the exploration policy. The exploration policy follows different criteria (for example the distance from the current agent position or the area that can be covered moving in a specified position) that can be formalized in multiple ways.
Many models that are proposed in the literature are based on the concept of *frontier exploration*: the agent, following different strategies, has to move to a *frontier* that is the region between free space and unexplored space. This family of exploration strategies can also be enhanced in different ways using Deep Learning.
Besides this famous implementation other methods are developed to select the next goal location that must be fed to the navigation policy module. Some of these methods are based on Deep Learning networks capable of predicting the next location to reach during exploration. Others are geometric methods that select the next point of interest with geometric heuristics.
We now describe how all the exploration policies described so far can be implemented.

Frontier exploration

Frontier exploration is one of the most popular exploration strategies, firstly introduced by [121]. It is based on the concept of frontiers, regions on the boundary between free space and unexplored space. The most common map used in this strategy is a grid-based map (Section 2.2) where each cell stores a value representing the probability of the corresponding location to be free or occupied by an obstacle. The frontier exploration approach maximizes map coverage by moving to new frontiers. At the beginning a prior probability value is assigned to each cell in the grid (usually 0.5, but different papers

propose a specific method to calculate this value). The cells are updated when the robot is moving, using the sensors readings. After the update the cells are placed into one of three classes, according to their value:

- **open:** occupancy probability $<$ prior probability.
- **occupied:** occupancy probability $>$ prior probability.
- **unknown:** occupancy probability $=$ prior probability.

Frontiers are so defined as a group of adjacent edge cells, where an edge cell is any open cell adjacent to an unknown cell. A threshold can be set to the minimum number of adjacent edges cells required to consider the group a frontier. An example can be seen in Figure 2.12.

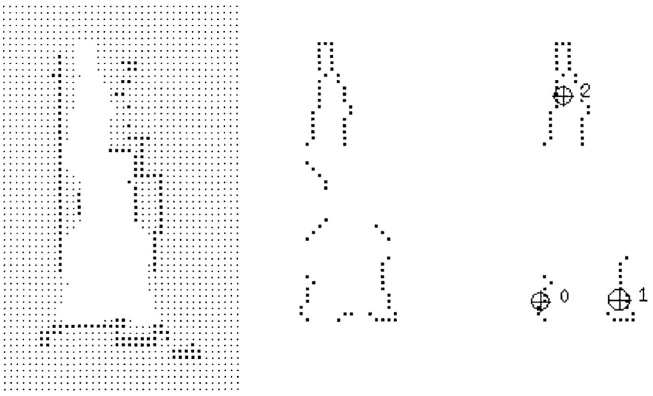


Figure 2.12: On the left an example of grid-based map used in frontier exploration; in the middle the frontiers extracted from the grid-based map; on the right frontier regions after threshold (right) [121].

The next frontier to be visited can be chosen in different ways, peculiar of each implementation, for example:

- **Random exploration:** the agent chooses a random element from the frontiers list.
- **Greedy selection:** the agent chooses as next exploration goal the point closest to the current position, as implemented in [66].
- **Next-Best-View selection:** the agent selects the frontier point that maximizes the amount of unknown area that can be seen from it, as shown in [43].

The point that must be reached is usually the point at the center of the frontier.
The maps used as input of this kind of algorithm can be of different types, as described in Section 2.4.2 without or with predicted elements, like in [100], [106], [124], [96], [66] and [82].

In [96] the authors use the predicted map to select the best frontier that is also closest to the point-goal for point-goal driven exploration. In [78] the authors use frontier exploration to solve object-goal driven exploration task, selecting the region that is more likely to contain the goal object (Deep Learning is used to create and update the semantic map).

Points of Interest (POI) and IDLE

In [60], the authors propose another method that doesn't rely on Deep Learning, but on geometric heuristic. In this method, the agent first needs to select *Points of Interest (POI)* from the updated map. They use two methods to extract POIs:

- POI is added to the selected list if a value difference between two sequential laser readings is larger than a threshold.
- POI is added to the selected list if sequential laser readings return a non-numerical value (reading out of range, represented as free-space).

If POIs are near an obstacle, they are removed from the memory. The optimal POI is selected with an evaluation method that the authors have developed, *Information-Based Distance Exploration (IDLE)*. The POI with the smallest IDLE is selected as the optimal waypoint provided as input to the navigation policy.

Integrating frontier-based exploration with Deep Learning

The frontier-based exploration algorithms can be integrated with Deep Learning techniques in different ways. For example, authors of [66] use a Deep Reinforcement Learning technique in order to learn how to select the best frontier point. They use A3C (asynchronous advantage actor-critic) network to maximize the total information gain along agent's navigation path.

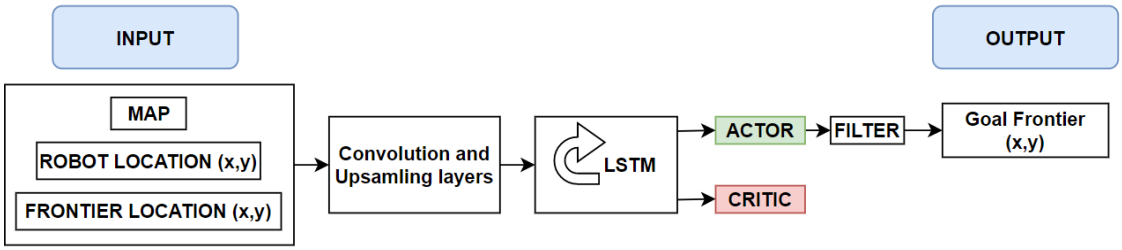


Figure 2.13: High-level schema of the Deep Learning network proposed in [92] to evaluate frontier points.

Figure 2.13 represents the network used in [92]. The network takes as input the current discovered map, the robot locations, and the frontiers location. These inputs, before be-

ing fed to the actor-critic network, pass through convolution layers, upsampling layers and a long short-term memory (LSTM) (this unit is used to make the network take into consideration previous robot state features). The network produces as output the goal frontier location that maximizes the objective function.

Selection of the next exploration goal using Deep Learning

In the examined papers different algorithms other than frontier exploration have been proposed to implement a policy capable of selecting the next point goal in the environment at each step. Many of these algorithms take advantage of Deep Learning networks like: CNN [36], actor-critic network [87], ResNet18 [67], Multilayer Perceptron (MLP) [95], Gated Recurrent Unit (GRU) [59]. Most of the examined policies are trained with Reinforcement Learning, while only one work uses Supervised Learning. The different implementation of Reinforcement Learning algorithms take into consideration different aspects of the exploration to reward.

The first example of algorithm trained with Reinforcement Learning is the one proposed in Figure 2.14.

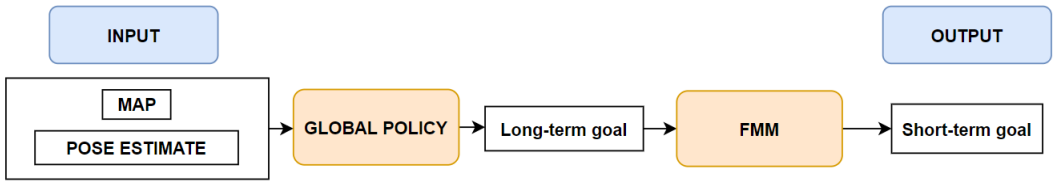


Figure 2.14: High-level schema of the Deep Reinforcement Learning network proposed in [48] to select the next exploration goal.

The figure is taken from [48], where the Global Policy uses a CNN. This network takes as input the updated map end the pose estimate and outputs a short-term goal. Then with Fast Marching Method (FMM) (described in Section 2.4.4) they compute the shortest path between the current position and the long-term goal: a short-term goal is selected along the path. Global policy is trained with Reinforcement Learning and the reward is proportional to the increase in area coverage.

Also in [97] the authors use a similar structure (trained with PPO), with some differences in the reward function. Reward function in this paper doesn't reward the amount of area seen, but the accuracy of the predicted map.

Also [49], [51], and [111] use a similar structure, but they eliminate the selection of the short-term goal with FMM, so the long-term goal is directly fed to the navigation policy. In particular, in [111], authors give a positive reward in case of increase explored area and a negative reward in case of collisions.

The work of [49] is trained with Reinforcement Learning (PPO) and it rewards the reduction in distance to the object-goal.
In [51], the exploration policy is likewise trained with RL (PPO) but authors introduce a new reward called *gainful curiosity* reward. With gainful curiosity reward the authors encourage the agent in the environment to find new objects and keep looking at them (from different points in space), until it obtains high confidence in predicting object categories.
In [52], authors try to accomplish the audio-goal driven exploration task with the structure shown in Figure 2.15:

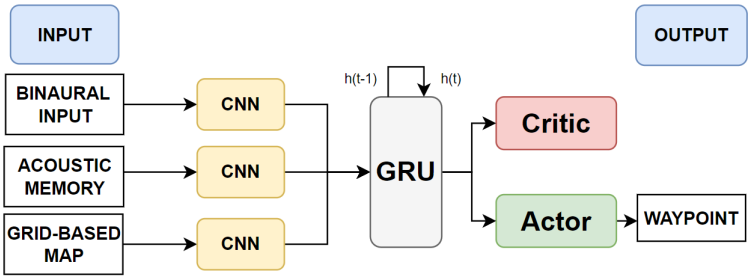


Figure 2.15: High-level schema of the Deep Reinforcement Learning network proposed in [52] to select the next exploration goal.

The network takes as input the binaural inputs, the acoustic memory, and the grid-based map. The three inputs are connected to a CNN and then concatenated into a GRU, connected to an actor-critic network that outputs the predicted waypoint that must be reached by the navigation policy. The policy is trained with Reinforcement Learning (PPO) also in this case and it is rewarded for reducing the distance to the goal.
An example of work that doesn't use Reinforcement Learning is the one proposed in [90] where authors use Supervised Learning to train a MLP network to output the exploration point that must be reached. MLP takes as input the predicted map, the current image seen by the agent (RGB), and the room ID where it is asked to navigate (it solves area-goal driven exploration tasks). RGB images and predicted maps inputs are embedded using a pretrained ResNet50 network.

2.4.4. Navigation policy

Navigation to the point selected by the exploration policy can be classified in two different ways: **classical path planning algorithms** and **learned policies**.

Classical path planning algorithms

In different papers authors use well-established numerical techniques to navigate to the point selected by the exploration policy. We have called them *classical* to distinguish them from those that use Deep Learning.
The most used are:

- **A* and Dijkstra:** these algorithms are used to plan a path from the current position to the point selected by the exploration policy in [111], [106], [96], [92], and [82].
- **Fast Marching Method (FMM)** [104] [34]: this algorithm is used for path planning in [49] and [51].

Dijkstra algorithm

Dijkstra algorithm is used to determine the shortest path between a start node and any other node in a graph. In path planning problems with nodes we refer to the points in space to be reached. The key idea of this algorithm is to iteratively calculate the shortest distance from a starting point, and can be described with the following steps:

1. Initialize all nodes with infinite distance; starting node is initialized with distance 0.
2. Only the distance of the starting node is marked *fixed*, all the others can change. All the nodes are marked as *not visited*.
3. Set the starting node as *active*.
4. Calculation of the temporary distances of the *not visited* neighbours of the active node.
5. If the temporary distance in the active node neighbours is lower than the current, update the distance and set active node as *predecessor* of the neighbour node.
6. Set the node with the minimum temporary distance as *active* and mark its distance as *fixed*. The previous *active* node is marked as *visited*.
7. Repeat from 4 to 7 until all nodes are visited.

This is the general formulation of the algorithm. It can also be adapted to find the

shortest path between a start and a goal point in space, by stopping the algorithm once the shortest path to the goal has been found.

A* algorithm

A* is a search algorithm used to find the shortest path between the initial and final node. This algorithm is widely used in various applications, including, of course, maps. The key part of this algorithm is the evaluation function $f(n) = g(n) + h(n)$, where:

- $g(n)$: is the accumulated cost to reach node n.
- $h(n)$: known as the *heuristic value*, it is the estimated cost to reach the goal from node n.

In path planning problems with node we refer to the points in space to be reached. Having defined these values, $f(n)$ represents the total estimated cost of a path passing through node n. At each step the algorithm moves to the node with the lowest $f(n)$ value, starting from the initial point. Different heuristics can be used to estimate the distance from one point to the goal, like the *Euclidean distance heuristic* or the *Manhattan distance heuristic* [105].

Fast Marching Method (FMM) algorithm

Fast Marching Method is a numerical algorithm originally thought for modelling and tracking the motion of a wave front. FMM has been applied to a lot of different fields, including, of course, path planning. The mathematical model of FMM can be applied to a wave in 2, 3, or n-dimensions. FMM calculates the time that a wave originated in one or more points needs to reach every point in the space. The motion of the wave front in every moment is described by the *Eikonal equation* [93]. The wave expansion speed is non-negative and can be different in different parts of the environment.

This model can be applied to path planning considering for example a grid-based map where free-space is valued as 1 and obstacles as 0. If our goal is to compute a path from a start point to a goal point, we could expand a wave from the starting point. The wave expansion speed is 0 at obstacles and 1 on free spaces. The path followed by the wave front from start to goal point, because of the wave expansion properties (expansion speed is considered constant), is the one with the shortest path.

Learned policies

In the literature several works use Deep Learning to learn a navigation policy in order to reach a point goal selected by an exploration policy previously defined. Authors use both pretrained networks and networks that need training. These policies can be formalized in different ways. In almost all the cases the policies are trained with Reinforcement

Learning, rewarding different aspects of the navigation and producing as output the next action to be executed. In only one of the examined cases authors train the policy with Imitation Learning and in another one with Self-Supervised Learning. Different Deep Learning networks are used: actor-critic [87], long short-term memory LSTM [70], and ResNet18 [67].
In [60] the exploration policy produces as output the description of the local environment (bagged laser readings in 180 degrees range in front of the robot) and the waypoint goal that the agent must reach. These outputs are merged together as input for the navigation network. The navigation network is an actor-critic network trained with Twin Delayed Deep Deterministic Policy Gradient (TD3) [63] to output an action, as shown in Figure 2.16.

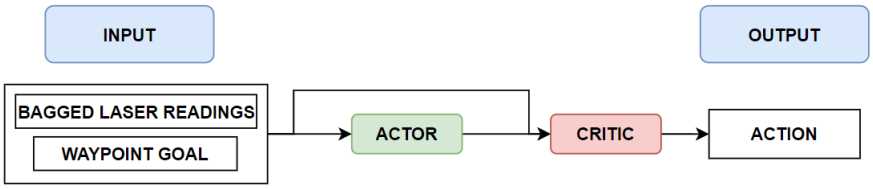


Figure 2.16: High-level schema of Deep Reinforcement Learning network proposed in [60] to select the next navigation action.

The policy is trained with Reinforcement Learning (TD3) and gives a positive reward if distance to the goal is less than a threshold, a negative reward in case of collision and if none of the previous conditions are satisfied it gives a reward based on the current linear velocity and angular velocity. The authors also employed a delayed attribute reward method, giving a positive reward not only when the goal is reached, but also decreasingly over the last steps before it.
Also in [123] authors use an actor-critic based PPO algorithm to train the Reinforcement Learning policy. In the paper they consider the case when the agent has to avoid a pedestrian crowd moving in the environment. The network takes as input the laser scans, the current velocity, and the goal provided by the exploration policy. The reward function is designed to encourage the robot to avoid collisions, reach the destination as fast as possible, and move the agent as smooth as possible (controlling its rotational velocity). The output is a collision-free velocity command (linear and angular).
In [90], the authors train the point navigation policy using Reinforcement Learning Proximal Policy Optimization (PPO) [103]. The policy, parametrized as a 2-layer LSTM network, takes as input the previous action, the goal predicted by the exploration policy and the encoding of the depth image input. It produces as output a softmax distribution over the action space and an estimate of the value function, in order to select an action

that must be executed. The reward is the reduction in distance to the goal.
In [97] the authors use a pretrained ResNet18 recurrent neural network, originally trained with Reinforcement Learning rewarding the reduction in distance to the local goal. The network takes as input the current RGB observation and the short-term goal selected by the exploration policy and outputs a navigational action.
The policy used in [48] is the only one trained with Imitation Learning (Behavioral Cloning). Also in this case the network is ResNet18, which takes as input the current RGB observation and the short-term goal selected by the exploration policy and outputs a navigational action.
In [101], the authors use a ResNet18 network as implemented in [18], but they train the network on the specific task instead of using a pretrained network. The network used in this paper takes as input two RGB images (current observation and goal observation) and outputs a softmax distribution of all the available actions. The policy is the only one trained with Self-Supervised Learning.

2.4.5. Exploration and navigation end-to-end learning

In different papers, exploration and navigation are jointly learned using and end-to-end paradigm and can't be clearly separated in two modules as in the previous cases.
In the following, we describe some algorithms proposed to implement end-to-end module.

Learning a policy with different Deep Supervised Learning networks to select next exploration action

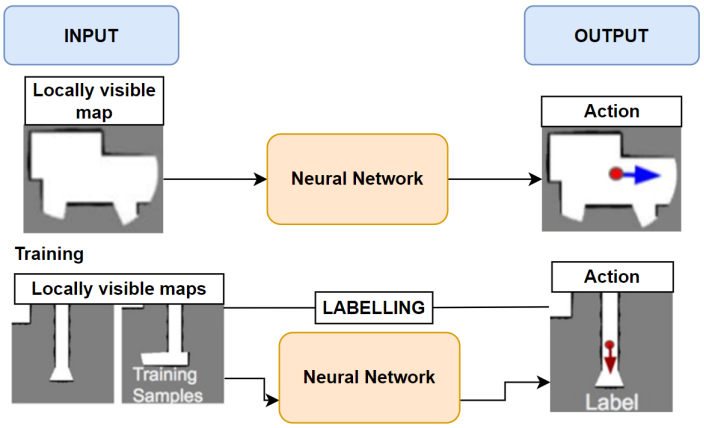


Figure 2.17: Pipeline of the end-to-end module presented in [41].

In [41], as shown in Figure 2.17, authors use a Deep Learning neural network that takes as

input the locally visible portion of the map and outputs the optimal sensing action. The network is trained and tested on locally visible portions of randomly-generated 2D maps of indoor environments. The authors have tested the algorithm with different neural networks: AlexNet [77], VGG [109], GoogleNet [114], ResNet [67], Locally Connected Layers [57].
The networks are trained with Supervised Learning, as shown in Figure 2.17. Labels are calculated offline before the training phase. Every label represents the index of the optimal action that maximize the agent's information gain (the action that allows the agents to collect as much information as possible).
The network proposed in this paper must be carefully considered, because it is the only one among all the examined papers that uses a Supervised Learning network structure. This method carries along some drawbacks, for example the need of labeling the actions before training. In addition, the method is only trained and tested on 2D randomly generated maps and not on 3D more complex environments.

Learning a policy with Reinforcement Learning and Imitation Learning to select the next exploration action

In [50], the authors use a network made of a pretrained ResNet18 visual encoder, followed by two fully connected layers and a GRU layer. The network takes as input a semantic map and outputs an action distribution and is trained with Reinforcement Learning (PPO). The authors define the *semantic curiosity reward*, a reward that encourages to explore frames with temporally inconsistent predictions (objects in the scene are labeled different in subsequent frames).
In [55] a similar pipeline is used, as shown in Figure 2.18.

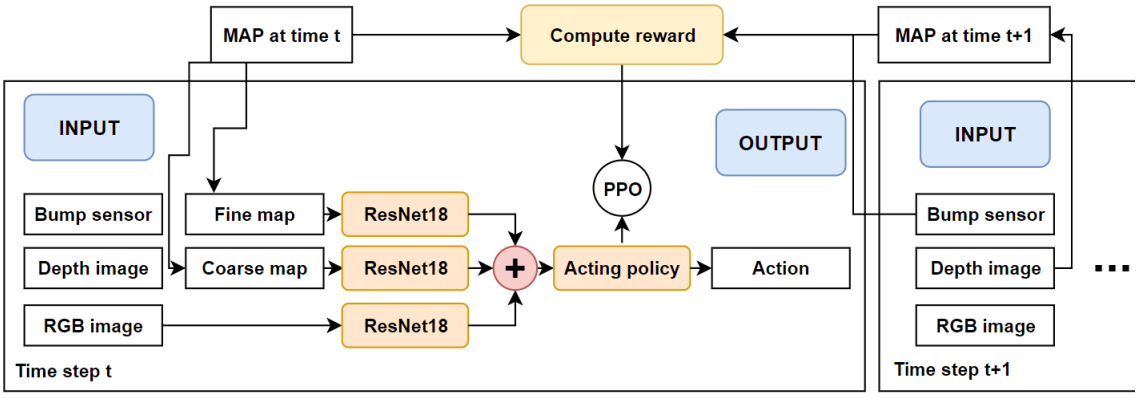


Figure 2.18: Pipeline of the end-to-end module presented in [55].

The network takes as input RGB images (processed with pretrained ResNet18), information from grid-based map (features extracted with ResNet18). The outputs of these

two ResNet18 networks are merged together into a Recurrent Neural Network in order to output the next action. Approximate map and bump sensor information are used to compute the reward function. Pretraining is done with Imitation Learning to imitate human demonstrations of how to explore a new environment. The demonstrations are composed of trajectories of real workers moving in House3D [62] environments.
Training is done with Reinforcement Learning (PPO). Reward function gives positive reward if the current observation adds obstacles or free-spaces to the map. It adds a negative reward if a collision (identified by the bump sensor) occurred.

Learning a policy with different Reinforcement Learning algorithms to select the next exploration direction

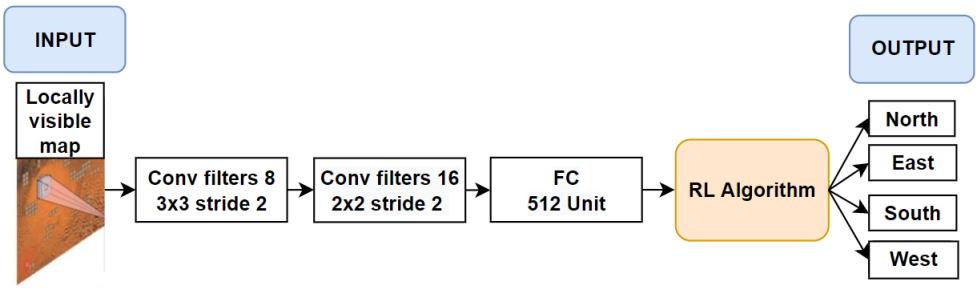


Figure 2.19: Pipeline of the end-to-end module presented in [76].

In [76], as shown in Figure 2.19, the authors use different Reinforcement Learning algorithms that output the direction that must be followed by the agent. The input is represented by the sensory information acquired by the robot in the environment.
The authors have tested different algorithms: PPO [103], DQN-Rainbow [69], A3C [86], SAC [65]. All the algorithms are trained with the same reward function that gives a positive reward if new cells are discovered, a negative reward per timestep (in order to avoid unnecessary movements), a negative reward if the action done leads to invalid states and a positive reward when more than a threshold value of the cells have been explored.

Learning two Reinforcement Learning policies simultaneously to select the next exploration direction in mapless exploration

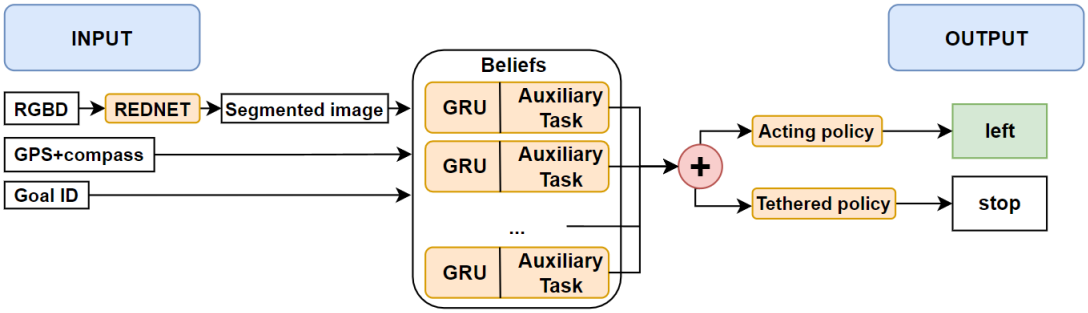


Figure 2.20: Pipeline of the end-to-end module presented in [122].

The work of [122] presents an architecture with different peculiarities. In almost all the examined papers authors use explicit map representation (grid-based maps in most of the cases), while in this work they don't store any expilict representation of a map. The inputs (RGBD images, GPS+Compass, and goal ID) are directly fed to the Deep Learning network for exploration policy and no map is created. We have called this situation mapless (as described in Section 2.2). Another example of mapless exploration is provided by [71], where aerial robots are implemented (in all the other papers examined only terrestrial robots are implemented).
Another distinctive element introduced by [122] is the fact that unlike the others presented in the thesis, it produces two outputs, as shown in Figure 2.20.
The network inputs go through belief modules, independent GRUs, each associated with independent auxiliary tasks. The belief modules output is merged and used in a linear actor-critic policy head. The agent can learn two policies simultaneously, for example the *acting policy* for the exploration for map buildingtask, and the *tethered policy* for object-goal driven exploration task.
Each policy is trained with Reinforcement Learning and has its own reward and the agent can act according to any mix of the two policies (because the two policies share the same action space).
In the paper, the acting policy for exploration is rewarded with a positive reward if goal is reached, a negative reward for each step, and a positive reward for area explored in the step. The tethered policy for object-goal driven exploration is only rewarded with a positive reward if goal is reached.

2.5. Simulation environments and datasets

Deep Learning techniques are characterized by a flexibility that made them suitable to be deployed in different applications like speech recognition, image recognition, natural language processing, and so on. These techniques, as known, are based on the concept of *training*. In the robotics field, as in the other fields, these means that the robot has to train a lot in order to obtain good performance. The training phase can't be done in the real world, but it has to be accomplished in a simulator. However, the simulation environments can be very different from real ones. Deep Learning algorithms working well in simulated environments may prove to be not so effective in the real world. This behavior is due to the fact that they learn specific features from the simulated environments that do not exist in the real world, assuming for example unrealistic agent motion not influenced by noise, perfect knowledge of its pose, or exploiting simulation behavior that can't happen in the real world. In Table 2.2 we have highlighted all the datasets (and the corresponding environment types) and the simulators used in the examined papers.

Paper	Dataset	Environment type	Simulator
[66]	HouseExpo [79] and generated	Indoor (floorplans)	Simulation in Python
[46]	KTH [40]	Indoor (KTH floorplans)	ROS stage [26]
[100]	Environments procedurally generated	Tunnels	Simulation in Python
[106]	KTH [40]	Indoor (floorplans)	Stage [117]
[48]	Gibson [10] and Matterport3D [20]	Indoor	Habitat simulator [4]
[74]	HouseExpo [79] and HOME [14]	Indoor (houses and Japanes Homes)	Gazebo [9]
[92]	Environments procedurally generated	Indoor/Outdoor (harsh scenarios)	ROS stage [26]
[90]	Matterport3D [20]	Indoor (house rooms)	Habitat simulator [4]
[75]	Google Cartographer	Indoor (buildings)	Gazebo [9]
[39]	KTH [40]	Most indoor, but no all	Gazebo [9]
[97]	Gibson [10] and Matterport3D [20]	Indoor (houses/offices)	Habitat simulator [4]
[110]	Stachnis (link not accessible)	Indoor but generalizable for tunnel, caves and mines	N.A.

[52]	Replica [28] and Matterport3D [20] with SoundSpaces audio	Indoor	Habitat simulator [4]
[82]	Ros stage environments	Indoor (map with walls needed, office and schools with more than 15 rooms)	ROS stage [26]
[124]	HouseExpo [79] and generated	Indoor (boundary known a priori)	Simulation in Python
[55]	SUNGC (not available)	Indoor (apartments)	House3D [119] based on SUNGC (not available)
[101]	Doom environments	Doom maze (videogame)	Vizdoom [33]
[41]	Environments procedurally generated	Dungeon	N.A.
[60]	Rooms randomly generated in gazebo	Indoor	Gazebo [9]
[76]	Environments procedurally generated	Outdoor (Mars environments)	Paper Open-AI gym compatible environment
[49]	Gibson [10] and Matterport3D [20]	Indoor	Habitat simulator [4]
[50]	Gibson [10] and Matterport3D [20]	Indoor	Habitat simulator [4]
[89]	AI2THOR environments	AI2THOR environments (only kitchen)	AI2THOR [1]
[122]	Matterport3D [20]	Indoor	Habitat simulator [4]
[111]	N.A.	2D grid maps	N.A.
[96]	Doom environments	Doom maze (videogame) with dynamics actors and hazards	ViZDoom [33]
[78]	Matterport3D [20]	Indoor	Habitat simulator [4]
[71]	N.A.	Offices and underground tunnels	Gazebo [9]
[51] [123]	Gibson [10]	Indoor Pedestrian Crowd	Habitat simulator [4]

Table 2.2: Simulators, datasets and environment types used in every work analyzed in this chapter.

Most of the algorithms are tested in environments where no other agents are moving. A notable difference is [96] that considers dynamic (hostile) actors; however the environments where the algorithm is trained are taken from Doom (the famous videogame). In Doom levels the other agents have scripted moveset and their predictability don't fully represent situations that can happen in the real world.
The work of [110] considers a situation where multiple agents are exploring the environment. This element, of course, adds another cause of difficulty. However, in their work, the robots can communicate with each other, a solution not always available in real world situations.
Only in [123] the authors consider a real world situation where the agent has to move in a pedestrian crowd (and avoid all the people moving).
Most of the papers focus on indoor environments like homes or offices without taking into consideration what is the performance when deployed in outdoor environments (or vice versa). Deep Learning algorithms that have learned how to explore caves or mine tunnels may not be so good when deployed in an office. In emergency situations (like search and rescue) environments become very different and unpredictable with respect to the ones considered in the papers. Also this situation should be considered with more attention.
When we look at the simulators used in order to train and test the algorithms some differences arise. Some of them, like Gazebo and Habitat simulator (better described in Sections 4.1.4 and 4.1.2), allow simulation in rich 3D environments and also physics is simulated. Other simulators like ROS Stage use 2D environments. Also in papers where the simulator is specifically written in order to test the algorithm (like [111] or [53]) the authors use 2D environments.
Different papers propose the use of procedurally generated environments. These environments are usually very simple with respect to the ones contained in Gibson, Matterport3D, Replica, or AI2THOR datasets. These datasets, instead, try to reproduce real indoor spaces (scanned in 3D and reconstructed) and AI2THOR also allows the interaction with more than 2000 unique objects.

2.5.1. Noise in simulation environments

Inputs obtained by the agents in the simulation are very often *noise free*, a situation that doesn't exist in reality. The agents trained in noise free environments could suffer from decreased performance when deployed in the real world.
The majority of the examined papers don't take into account this behavior, but some of them do.
In [55], [97] authors try to improve the simulation physical realism by using noise models

on both actuation and odometry sensors in the Habitat simulator. Noise models are collected by letting the agent move in the real world and building the motion and sensor noise models. The two noise models are:

- **Actuation noise:** is measured as the difference between the real agent pose in the world after an action and the corresponding intended pose.
- **Sensor noise:** is measured as the difference between the sensor pose estimate and the real agent pose in the world.

The real robot used by the authors to collect the data is LoCoBot [19] and pyrobot API [88] together with ROS [23] are used to obtain the readings associated to the control commands fed to the robot itself. In their proposed algorithm the robot can be controlled with three navigational actions: forward, turn right, and turn left. For each action the authors fit two Gaussian mixture models: one for the actuation noise and one for the sensor noise. In order to get an accurate agent pose the authors use Hokuyo UST-10LX Scanning Laser Rangefinder (LiDAR). They use noise s for both training and evaluation. A similar model is used also in [89].

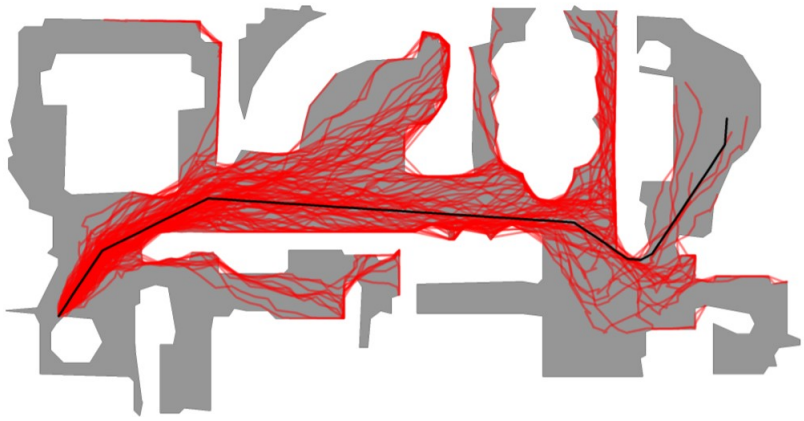


Figure 2.21: Actuation noise in Habitat simulator [15].

In Figure 2.21, from [15], we can see the effect of actuation noise implemented in Habitat simulator. The black line is the trajectory that represents the action sequence in an environment with no noise and perfect actuation. The red lines represent the trajectories obtained from the black line with actuation noise. The black line represents the behavior followed by the robot acting deterministically (for example when the agent executes turn right 15 and it turns exactly 15). No robot moves deterministically in a real world environment and this behavior is represented by the red lines. The image represents 100 possible trajectories for the deterministic trajectory. The figure shows how the same ac-

tions can lead to very different final locations when we consider noise.
In [73], the authors test what is the influence of noise in training on Habitat simulator. The authors test the agent in point-goal driven exploration task in a simple real world environment (a room with some boxes/obstacles). They have tested different algorithm trained in Habitat environment with noise settings and they have discovered that the best performance in the real world was obtained when no actuation noise was involved during training. This behavior may suggest that the sampled noise doesn't correctly reflect the situations that we can have in a real room.

2.6. Algorithms tested on robots in real world

As already said, Deep Learning exploration algorithms can exploit imperfections to obtain an exploration behavior that can't really be replicated in the real world. In a simulator, an agent may learn to move in ways that can't be replicated in reality. For example, as reported in [16], agents in Habitat simulator used to slide along obstacles when colliding, leading to paths that can't be followed in the real world. As reported in [73] the *sliding behavior* can be found in other simulators, like Gibson, AI2THOR, MINOS [102], Deepmind Lab [42], and in different video game engines. Because of these unintended behaviors it is important to test exploration algorithms in the real world in order to fully understand their capabilities. Only a few number of the algorithms presented in the analyzed papers are also tested on a real robot (e.g., [48] and [49] on a Locobot, [110] and [60] on a Pioneer P3-DX, [92] on Turtlebot 2, [71] on DJI Snail racing drone). In [73] the authors have tested an algorithm trained in Habitat simulator in a real environment for the point-goal driven exploration task. The authors have 3D-scanned a lab space to create a virtualized replica and tested the algorithms in both the simulation environment and the real world. The lab is a very simple environment when compared to the ones used in Habitat simulator datasets but even in a simpler environment algorithm has lower performance. They have also discovered that some algorithms performing better than others in simulation environments, weren't the best in the real world environment.

Figure 2.22: Reality on the left and simulation environment on the right (from [73]).

3 | Problem definition

In this chapter, we describe the problem analyzed and discussed in the thesis work, i.e., what are the advantages and disadvantages of Deep Learning algorithms over methods commonly used in the past to solve the exploration problem. In particular, Section 3.1 describes the exploration problem, while Section 3.2 describes the goal of this thesis. Sections 3.3 and 3.4 describe what are the algorithms chosen to perform the comparison between traditional and Deep Learning algorithms, while Section 3.5 describes the simulator used. Finally, Section 3.6 illustrates how the comparison should be implemented.

3.1. Exploration problem

Autonomous exploration is an important task that a robot deployed in an unknown environment must accomplish [47]. The robot with no previous information about the environment has to choose where to move and consequently the best strategy to explore the environment in order to build its map incrementally. Over the years different strategies have been proposed and developed. Some algorithms proposed to solve the exploration problem are described in Section 2.4.
In the last few years, Deep Learning techniques have become very popular in different fields like speech recognition and image or video recognition. Researchers have therefore decided to use Deep Learning and in particular Deep Reinforcement Learning to solve the exploration problem, in contrast to methods commonly used in the past, that we consider as *classical*. Different examples are reported in Section 2.4.
In the same section, we have described how the exploration steps can be divided in different modules in order to understand where Deep Learning or classical (like frontier exploration) techniques can be used. The schematization is the one proposed in Figure 3.1.

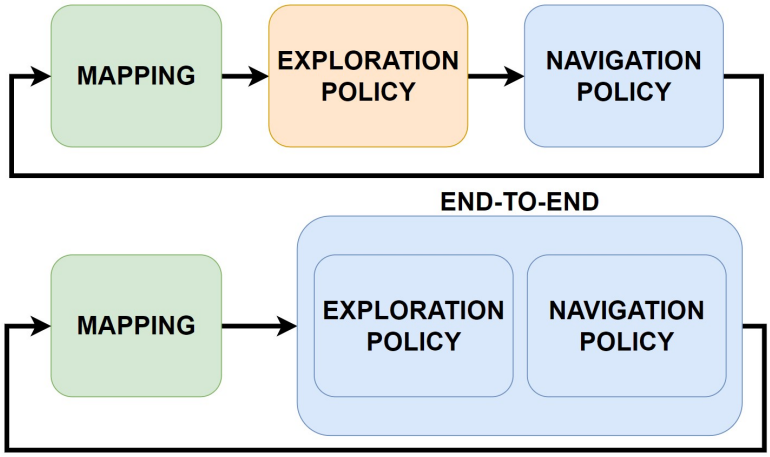


Figure 3.1: Schematization of the exploration problem. On the top the case where exploration and navigation policies are considered separate modules. On the bottom the case where exploration and navigation policies are considered a single module, learned end-to-end.

From Figure 3.1 we can see the different modules where classical or Deep Learning techniques can be used in the exploration framework: **mapping**, **exploration policy**, and **navigation policy**.
Mapping is the module in charge of generating or updating the map. The map can be simply updated with the information coming from the current observations or enhanced with predictions. Exploration policy module is responsible for the selection of the next point that must be reached during the exploration and navigation policy is the module responsible for reaching that point. Exploration and navigation can also be solved end-to-end with Deep Learning algorithms. As described in Section 2.3 there exist different exploration tasks that can be solved by the algorithms and the most common are exploration for map building and point-goal driven exploration.

3.2. Purpose of the thesis

The purpose of this thesis is to experimentally compare classical exploration algorithms (like frontier exploration) and Deep Learning algorithms in order to understand what are the positive and negative sides of the different techniques.
The community of researchers that is dedicated to study classical exploration algorithms and the one that devotes its time to Deep Learning exploration paradigm are in most of the cases separate communities. Many researchers in the field of classical exploration rely on these techniques because they have proven to be robust and with reliable performance

also when used by actual robots in the real world; however, this may prevent them to benefit from innovations and better performance that may be offered by Machine Learning and in particular Deep Learning. Deep Learning researchers, on the other hand, in most of the cases compare algorithms only with other learning-based approaches, without taking into account classical exploration algorithms and the challenge of deploying the agents in the real world.
The classical methods that can be used in navigation module (A* and Dijkstra) already offer the optimal solution and because of this fact we have to understand if Deep Learning algorithms are worth using in this module. The classical methods used in mapping or exploration module can have advantages or disadvantages depending on the implementations and because of this we have to understand if Deep Learning techniques can offer better results.
The first step in order to compare these paradigms is to find algorithms that implement Deep Learning techniques for the different modules described in Figure 3.1.
The next step after having identified the algorithms to compare is to select the right simulation environment. As simulators use very different environments when compared to the real world, Deep Learning algorithms may learn how to exploit fictional features to solve their tasks, features that may be unique to the simulation but that don't really exist in a real-world deployment. One example of this behavior is reported in [17]. Here the author reported an experiment done at the University of Washington to create a classifier capable of distinguishing between images of wolves from huskies. The system managed to obtain 90% accuracy, but with later experiments, they discovered that the model was basing its decisions on the background. Wolf images usually had snow in the background, while husky images usually did not. The researchers have so created a snow detector, rather than a husky/wolf classifier.
Classical and Deep Learning algorithms must be tested in the same environments on common metrics in order to understand what are the differences in terms of performance, reliability, and generalization.
Deep Reinforcement Learning algorithms tested in very simple environments (small or few rooms with few objects in them) must also be tested in more complex environments. This test is useful to understand the solidity of the reward function and its ability to generalize. In fact, as described in Section 2.4.1, writing a reward function can be a difficult task and a reward function perfectly working in an environment may be not good in an environment with different characteristics.
Deep Learning algorithms that try to solve more than one of the different exploration tasks must be compared in all of them, in order to understand what are the strengths and limitations of the specific techniques in the different tasks.

3.3. Classical exploration algorithm

In order to compare Deep Learning algorithms on exploration for map building task with a classical exploration algorithm, we have selected as an example of a classical method the greedy frontier exploration of [8]. In greedy frontier exploration, as described in Section 2.4.3, the agent chooses as the next exploration goal the point, from the frontier list, closest to the actual position. More technical information is reported in Sections 4.1.1 and 4.2.2.
The classical algorithm used for comparison in this thesis is simple, but other more complex and efficient algorithms exist in the literature [37].
Figure 3.2 represents the classical structure selected to be compared with Deep Learning algorithms. The comparison on point-goal driven exploration task is done with a similar structure, but only with mapping and navigation module.

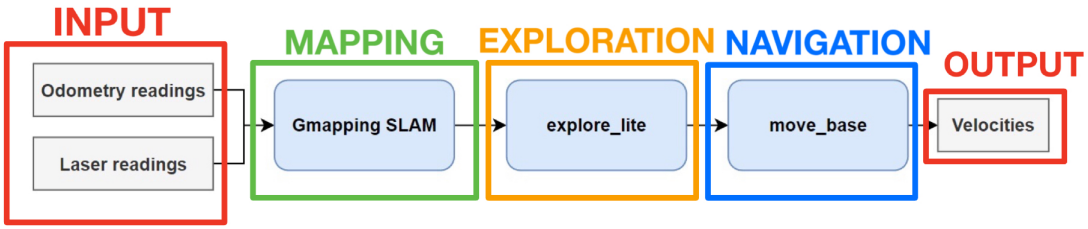


Figure 3.2: High-level schema of how the different modules of the exploration framework are implemented in classical frontier exploration algorithm.

Gmapping, explore_lite, and move_base are packages (modules that provide functionalities in an easy-to-consume manner [25]) provided by ROS, an open-source, meta-operating system for robots (better described in 4.1.1).

- **Input:** laser readings and odometry readings.
- **Modules output:** mapping module outputs a grid-based map; exploration module outputs the frontier point to be reached; navigation module outputs the stream of velocity commands the robot has to execute to reach the selected point.
- **Exploration tasks:** exploration for map building and point-goal driven exploration (only Gmapping SLAM and move_base module).
- **Mapping module:** the map is updated using SLAM procedure with Gmapping [12].
- **Exploration policy module:** greedy frontier exploration with explore lite [8].
- **Navigation policy module:** move_base [21] with Dijkstra for path selection.

3.4. Deep Learning algorithms - ANS, OccAnt, and DRL

In the list of algorithms proposed by authors of the papers analyzed in Chapter 2, only few present a public available code implementation. Among them, we have selected three algorithms that show different ways of implementing the three fundamental modules identified.

- **ANS** from D.S. Chaplot et al. [48].
- **OccAnt** from S.K. Ramakrishnan et al. [97].
- **DRL** from R. Cimurs et al. [60].

In the next sections, we provide a detailed description of these three methods.
ANS, OccAnt, and DRL present three different ways of implementing the mapping module. Whether both OccAnt and ANS use Deep Learning, DRL doesn't implement this technique in the module. In ANS the mapping module is trained to generate an estimate of only the visible occupancy that can be obtained from agent's egocentric view. In OccAnt the authors train the network to anticipate the area not directly visible because of occlusion (for example the area that can be around a corner or behind a wall or table).
Exploration policies in both ANS and OccAnt are trained with Reinforcement Learning, but they reward different aspects of the exploration. DRL doesn't use Deep Learning in the exploration policy module, but a geometrical method to select and evaluate POIs (Points of Interest), similarly to frontier exploration algorithm.
Navigation policy in all the three papers is trained with Deep Learning (but with multiple techniques).
ANS and OccAnt are developed in order to solve both exploration and point-goal driven exploration tasks, while DRL only solves point-goal driven exploration task.
Other works from the literature, like the one of [55], with end-to-end implementation, are not immediately replicable due to the fact that they do not provide source code, trained model, nor dataset to replicate their results.

3.4.1. Learning to explore using active neural SLAM - ANS

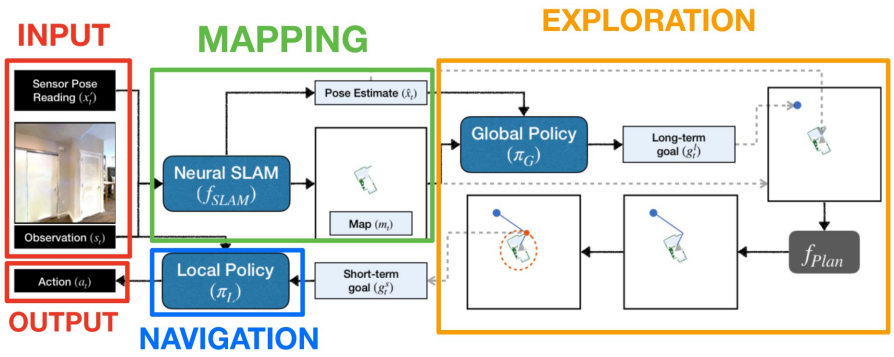


Figure 3.3: High-level schema of how the different modules of the exploration framework are implemented in ANS algorithm from [48].

- **Input:** sensor pose readings and observations (RGB images).
- **Modules output:** mapping module outputs a grid-based map and agent's pose estimate; exploration module outputs the short-term goal to be reached; navigation module outputs the action the robot has to execute to reach the selected point.
- **Exploration tasks:** exploration for map building and point-goal driven exploration.
- **Mapping module:** *Neural SLAM* module. This module takes as input the current RGB observation, the current and last sensor reading of the agent pose, the last map estimates, and the last agent pose. It outputs the updated map and the current agent pose estimate. Two learned submodules are used: the *map* module (that outputs the obstacles and explored area for the current observation) and the *pose estimate* module (that outputs the estimate agent pose taking as input the past pose estimate and last two map updates).
- **Exploration policy module:** the module used to solve exploration for map building task is made of two submodules: *global policy* and *fplan*. Global policy module takes as input the map, the pose estimated by the Neural SLAM module, and the visited locations and uses a CNN to output the long-term goal. The policy is trained using Reinforcement Learning (PPO) and rewards increasing area coverage. Fplan is a planner and takes as input the long-term goal, the obstacle map and the agent pose to output the short-term goal using the Fast Marching method.
  For the point-goal driven exploration task the module is the same, but global policy always outputs the point-goal coordinates and no additional training is needed.

- **Navigation policy module:** it is a Recurrent Neural Network (RNN) that uses pretrained ResNet18 as encoder. It takes as input the observations (RGB images) and the short-term goal produced by the exploration policy module. The policy is trained using Imitation Learning (Behavioral Cloning) to output a navigational action.

3.4.2. Occupancy anticipation for efficient exploration and navigation - OccAnt

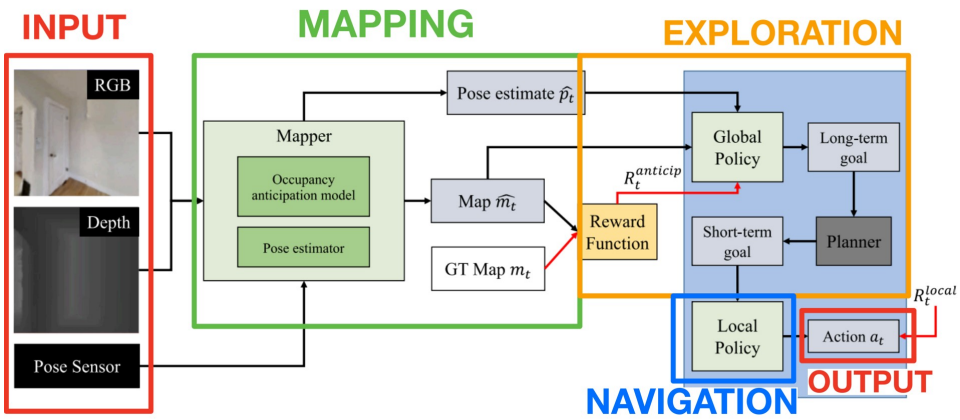


Figure 3.4: High-level schema of how the different modules of the exploration framework are implemented in OccAnt algorithm from [97].

- **Input:** RGB images, depth images, and sensor pose reading.
- **Modules output:** mapping module outputs a grid-based map and agent's pose estimate; exploration module outputs the short-term goal to be reached; navigation module outputs the action the robot has to execute to reach the selected point.
- **Exploration tasks:** exploration for map building and point-goal driven exploration.
- **Mapping module:** it is a U-net network that takes as input RGB and depth image and outputs the anticipated occupancy for the region in front of the agent.
- **Exploration policy module:** the module used to solve exploration for map building task is the same network as that of Section 3.4.1 trained with Reinforcement Learning (PPO) with a small difference; while ANS rewards increase in area coverage, in OccAnt the reward function rewards actions that allow to predict correctly the map, irrespective of the fact that the robot has actually observed that predicted location or not.

- **Navigation policy module:** an RNN trained with Imitation Learning, as in Section 3.4.1.

3.4.3. Goal-driven autonomous exploration through Deep Reinforcement Learning - DRL

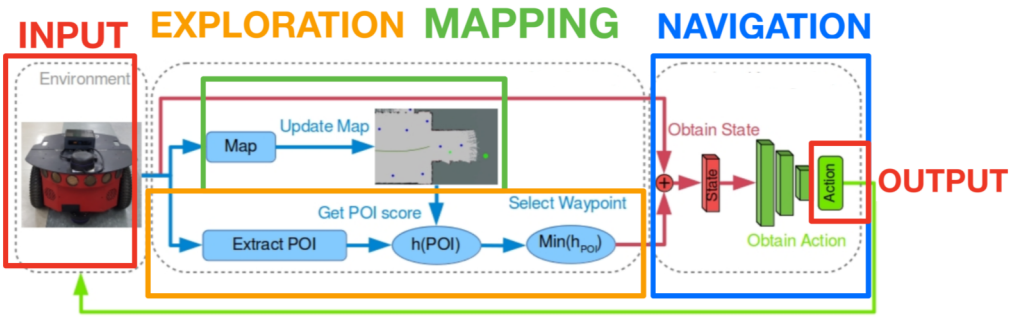


Figure 3.5: High-level schema of how the different modules of the exploration framework are implemented in DRL algorithm from [60].

- **Input:** laser readings and odometry readings.
- **Output:** mapping module outputs a grid-based map; exploration module outputs the point in space to reach; navigation module outputs the action the robot has to execute to reach the selected point.
- **Exploration tasks:** point-goal driven exploration.
- **Mapping module:** the map is updated using traditional SLAM procedure (in the simulator, ROS package SLAM Toolbox is used [83]).
- **Exploration policy module:** the algorithm at each step selects a list of *POIs* (Points of Interest). POIs are extracted with two methods:
  - POI is added if a value difference between two sequential laser readings is larger than a threshold,
  - POI is placed in the environment if sequential laser readings return a non-numerical value (reading out of range, represented as free-space).

  The optimal POI is selected with an evaluation method developed by the authors of the algorithm, *Information-Based Distance Exploration (IDLE)*. In IDLE authors evaluate a fitness function that considers the Euclidean distance between the agent and the POI, the Euclidean distance between the POI and the global goal, and a

map information score (that calculates the information around POI coordinates). The POI with the smallest IDLE is selected as the optimal waypoint provided as input to the navigation policy.

- **Navigation policy module:** is an actor-critic network trained with Twin Delayed Deep Deterministic Policy Gradient (TD3) [63] that takes as input the waypoint goal selected in the exploration policy and the description of the local environment (bagged laser readings in 180 degrees range in front of the robot). The network outputs a navigational action.

3.5. Simulation environment for comparison

In order to obtain a good comparison of the different algorithms mentioned so far it is important to choose a good simulation environment [38].
DRL is originally trained in the Gazebo simulator. Both DRL and frontier exploration are developed using ROS.
OccAnt and ANS are trained and tested in the Habitat simulator (more technical info about this simulator are reported in Section 4.1.4), with two different datasets that can be used: Gibson and Matterport3D.

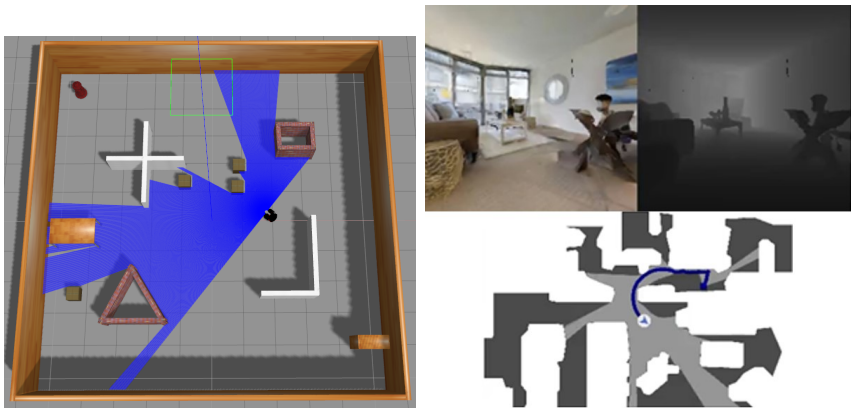


Figure 3.6: On the left an example of robot in the Gazebo simulator, on the right an example of robot in Habitat simulator with an environment from the Gibson dataset.

OccAnt and ANS code implementation are strongly dependent on the tools provided by the Habitat simulator and because of this cannot be tested on other simulation environments. As a consequence, we rely on the Habitat simulator as the main simulator used in the comparison.
DRL and frontier exploration can be used in the Habitat simulator using ROS-X-Habitat,

a software interface able to bridge the Habitat simulator with other robotics resources which use ROS (more information about this implementation is reported in Sections 4.1.4 and 4.2.2). Habitat is a physics-enabled 3D simulator specifically developed to allow fast simulation for Deep Learning algorithms that need a lot of training. Thus, Habitat is a good simulator choice because it allows algorithms to be tested with different photo-realistic datasets. All the photorealistic environments used during the comparison are assumed to be static, meaning that in the environments no any other agent is moving and the environment itself doesn't change during the exploration. The environments used are taken from the Gibson dataset, a reproduction of real indoor spaces made with 3D scanning and reconstructions. Frontier exploration, for a reason that emerged during the execution of the comparison, is also tested in the Stage simulator in a 2D environment. This has been requested because frontier exploration in the Habitat simulator with ROS-X-Habitat suffers from problems related to incorrect laser readings, because of holes in Gibson dataset textures. OccAnt and ANS don't suffer from this problem because they don't use the laser sensor, but only RGB and depth sensors. In addition, the agent gets stuck in invisible textures in all the exploration episodes. Using 2D maps of the same Gibson environments a more qualitative analysis can be obtained because no holes and invisible textures are present.

3.6. Key elements of the comparison

Different elements of the selected algorithms must be taken into consideration in order to obtain a fair comparison of the different exploration tasks.
OccAnt and ANS algorithms present a very similar structure. One of the most significative differences is about how the mapping module is implemented. While in ANS the mapping module simply integrates current agent observation in the map, OccAnt algorithm introduces an occupancy anticipation model used to predict occupancy in not visible areas. Because each one of these two algorithms is designed to solve both exploration and point-goal driven exploration tasks, it is important to understand if the occupancy anticipation model introduced by OccAnt is useful in both the tasks or in either of the two.
OccAnt, ANS, and frontier exploration can be compared on the exploration for map building task, in order to understand what are the downsides and upsides aspects of the Deep Learning and classical implementations.
In order to test the strength of the reward function written by the authors of DRL [60] and its ability to generalize, it can be useful to test the algorithm trained in Gazebo (DRL) also in Habitat more complex environments.
OccAnt, ANS, and DRL can be compared also with a classical algorithm on point-goal

driven exploration task.
In all the cases analyzed so far, the comparison on exploration for map building task must be done with the same starting points and the comparison on point-goal driven exploration task with the same starting points and goal points. Exploration and point-goal driven exploration are tested with specific metrics which depend on the particular task that must be solved and that will be discussed in Chapter 5.

4 | Implementation

In this chapter, we are going to describe the main software used during the implementation of the thesis work.
In Section 4.1, we describe the most important software components used in this thesis.
In Section 4.2, we describe how these software components are used in the different algorithms analyzed in this thesis.

4.1. Software components

In this section, we describe all the main pieces of software used in the thesis work.
This software is:

- **Robot Operating System (ROS):** an open-source, meta-operating system for robots [23].
- **Gazebo:** a simulator widely used in the robotic field, adopted together with ROS [9].
- **Stage:** a 2(.5)D robotics standalone simulator, usually adopted together with ROS [117].
- **AI Habitat:** a simulation platform developed to speed up research in the *embodied AI field* [84]. It works by enabling the training of *embodied AI agents* (intelligent agents with a physical body used to interact with the environment) in a photorealistic and efficient 3D simulator, in order to transfer the learned skills to the real world [115] [2].
- **ROS-X-Habitat:** a software interface that bridges the AI Habitat platform with other robotic resources implemented under ROS [54].
- **Utilities for Gibson environment**: a Python library [11] that offers a series of utilities for the Gibson environments described in Section 3.5.

4.1.1. Robot Operating System (ROS)

ROS is an open-source, meta-operating system for robots [23]. It provides the services we would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS helps in building robot components, like actuators, sensors, and control systems, and connects them using ROS tools.

Terminology

The most important elements of ROS are: *nodes*, *messages*, *topics*, *services*, and *ROS packages*.

A node is a process that performs computation [24] and it is used to divide the computation into a set of single elements. For example, one node can control the laser, another the robot's wheel motors, and so on. Nodes can communicate with other nodes with messages over the ROS network. Every node is a separated entity and it can be stored on one or more computers.

The messages are defined by their data format and types.

The communication network follows a classical publish/subscribe pattern. If a node wants to send a message (for example, the velocity values) it has to publish the message to a specific topic (in this case, in the topic called /cmd_vel). If a node wants to receive data from another node it has to subscribe to the specific topic published by the other node. A service is a synchronous remote procedure call (like the client/server mechanism): using a service a node can call a function on another node.

ROS master is a particular node that provides registration and naming services for the rest of the nodes in the network. It is responsible for tracking the publishers and the subscribers of all the topics in the network.

ROS organizes the software in packages, called rospackages, described by an XML file. A package might be made of nodes, libraries, datasets, third-party software, or other elements.

ROS also provides different helpful tools, for example, *rqt_graph*, a tool used to display a visual graph of all the processes (nodes, messages, ...) running in ROS and their connections. Another helpful tool is *RVIZ plugin*, which allows 3D visualization in ROS. It is used in [60] to display the map while training and evaluating the algorithm [27].

4.1.2. Gazebo

Gazebo is an open-source 3D robotics simulator. It allows to test algorithms, design robots, and train AI algorithms using different scenarios. Gazebo is highly supported by the community and it provides standard compatibility with ROS.
Gazebo allows programmers to create 3D simulated settings with robots, obstacles, and different kinds of objects. Gazebo comes also with a physical engine able to simulate gravity and inertia.

4.1.3. Stage

Stage is a 2(.5)D robotics standalone simulator that supports virtual worlds populated by different numbers of robots and sensors. Robots operate in a 2(.5)D bitmapped environment. Stage was originally designed to support both single-agent and multi-agent systems. Because of this is not intended to provide high-fidelity robot simulations,but good-enough fidelity, meaning that Stage is realistic enough to enable users to move robots developed in Stage simulator to the real world. Every robot can be equipped with different sensor and actuators models.

4.1.4. AI Habitat

AI Habitat is a simulation platform developed to speed up research in the Embodied AI field.

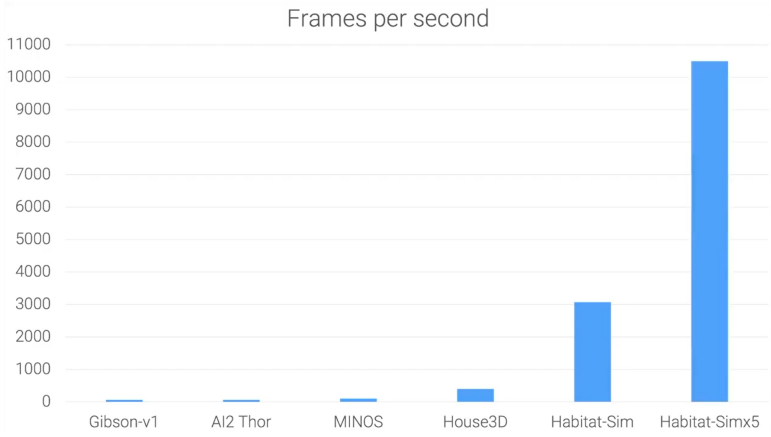


Figure 4.1: The frame-per-second in simulations for Habitat ([5]) when compared to other popular simulators.

The main goal of Habitat, as shown in Figure 4.1, is provide fast simulations with a higher

framerate. This capability to perform high framerate simulations is an enabling factor for many experiments that involve several training epochs/run and that thus were not very practical to carry out before.

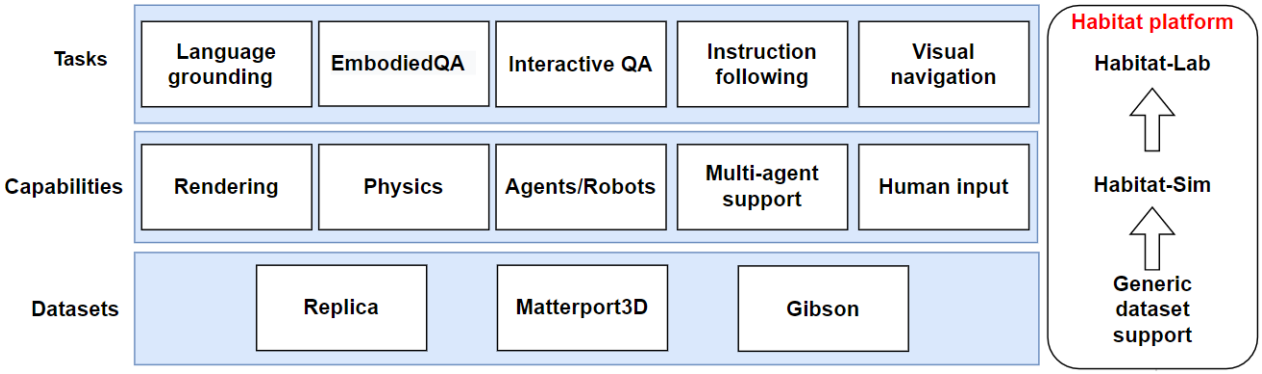


Figure 4.2: High-level schema of Habitat platform, from [84].

As shown in Figure 4.2, AI Habitat aims to standardize the *embodied AI software stack*. The embodied AI software stack used to train embodied agents is usually made of datasets (which provide 3D models), simulators that implement different capabilities, and the tasks that the agent has to accomplish.
Habitat proposes a standardization of this software stack based on a general flexible design capable of allowing people to load any datasets, to have a highly efficient implementation of different capabilities with a performant simulator (Habitat-Sim) and to define a variety of tasks with flexible APIs (Habitat-Lab). Habitat consists of two different modules:

- **Habitat-Sim** [4]: the physics-enabled 3D simulator itself. It supports:
  - 3D scans of indoor/outdoor spaces
  - Models of spaces and piecewise-rigid objects
  - Configurable sensors (RGBD cameras, egomotion sensing)
  - Robots described via Unified Robot Description Format (URDF)
  - Rigid-body mechanics
- **Habitat-Lab** [3]: a modular high-level library for end-to-end development. It allows to define the tasks, train the agents (with Reinforcement Learning, Imitation Learning, or no learning at all), configure the agents, and benchmark the performance on the defined tasks with standard metrics. It uses Habit-Sim as the basic simulator, but in principle, it can be used with any other simulator.

The photorealistic datasets most used with the Habitat simulator are: HM3D [13], Matterport3D [20], Gibson [10], and Replica [28].

Terminology

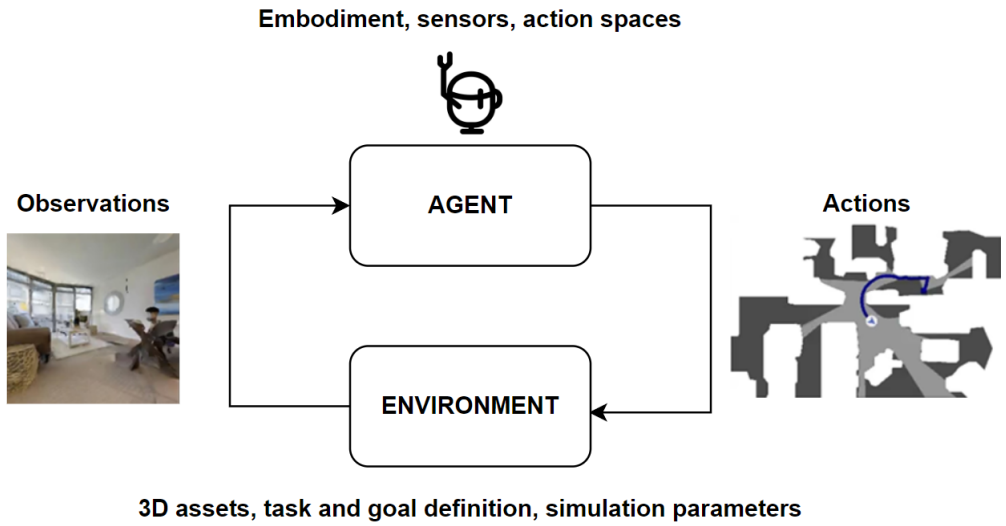


Figure 4.3: Habitat terminology, from [5].

The *agent* can take *actions* in a particular *environment*, and it receives from that environment some *observations*. From the observations, it can learn the current state of the environment and its state within that environment. The agent is associated with the concepts of embodiment, sensors, and action spaces. The environment is instead associated with the concepts of 3D assets, tasks, goal definition, and simulation parameters.

4.1.5. ROS-X-Habitat

ROS-X-Habitat is a software interface that is capable of bridging the AI Habitat platform with other robotics resources developed in ROS. Using this interface, people can train the Habitat Reinforcement Learning agent in other simulation environments to gain better generalization. It also allows using classical agents implemented with ROS packages in the Habitat simulator, as can be seen from the overview of the software interface reported in Figure 4.4. The reason why the authors proposed this framework is to overcome ROS and Gazebo (the standard simulation environment for ROS) limitations. ROS is one of the most popular tools in the robotics community, but its ability to directly support a Reinforcement Learning agent is limited. Gazebo simulator doesn't match the simulation speed of software specifically designed to support Reinforcement Learning training. ROS-X-Habitat, using an agent implemented with ROS packages, can also decrease the time required to move the agent to the real world (because of ROS support to real robots).

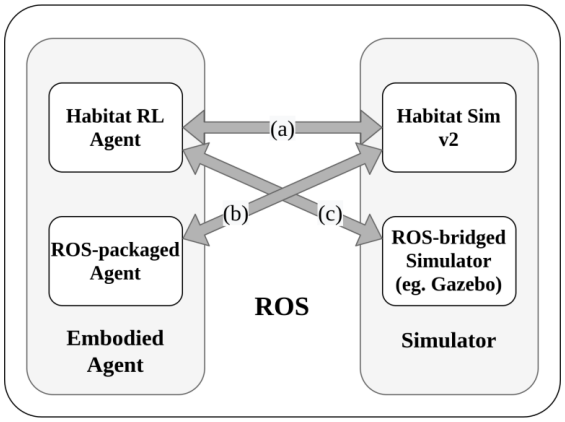


Figure 4.4: Overview of ROS-X-Habitat, from [29].

4.1.6. Utilities for Gibson Environments

This is a Python library that offers a series of utilities for Gibson environments described in Section 3.5. We have used this library to generate 2D floor maps (and their metadata) starting from Gibson *.obj datataset 3D files. The map's metadata created by this utility includes the origin's coordinate in pixel and the scale which indicates the real distance covered by a pixel. The produced 2D maps are used in Stage simulator.

4.2. Software implementation and changes

In this section, we explain how the software components presented in Section 4.1 are used in the different exploration algorithms considered.

4.2.1. DRL

DRL algorithm, from [60], is implemented by [7]. The authors simulate the robot in the Gazebo simulator, and the system is based on ROS. The proposed implementation doesn't directly consider any metrics related to the point-goal driven exploration task (like the number of successes during the episodes, the number of steps, or the distance to the goal at the end of the episode) and the metrics considered (average reward and average collision) are not directly produced as output. Because of this, we have added them to the code:

- Average reward and average collision over a number of episodes for every epoch
- Number of successes
- Total distance to goal in the comparison episodes
- Total number of steps in the comparison episodes

In order to test this algorithm in the Habitat simulator, we have adapted the code with the help of the utilities provided by ROS_X_Habitat. The architecture used is the one proposed in Figure 4.5.

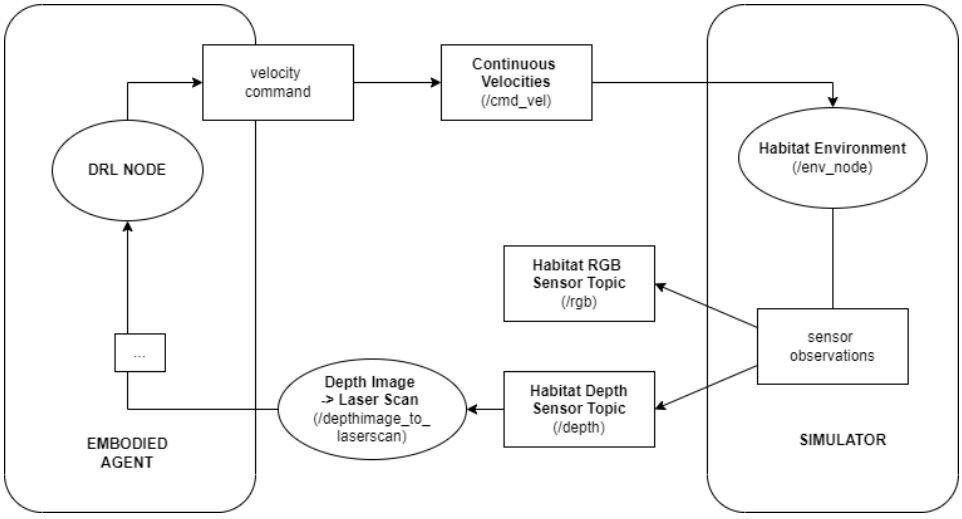


Figure 4.5: Schema of DRL implementation with ROS-X-Habitat [29].

With this architecture, the robot controlled by the DRL node is able to explore the

Habitat environment publishing ROS-based velocity command messages. The RGB and depth sensors readings are published as rgb and depth messages. Depth readings are converted to laser scan readings by using an ad hoc node before being processed by the planner. RGB readings are used only for representation purposes. An important note is about the fact that the Habitat simulator doesn't provide the implementation of the velodyne laser of Gazebo as used in [7]. The DRL developers provided the possibility to select different reward functions: some based on the value read by the velodyne laser and some not. Because of this possibility we have selected a reward function that is not based on the velodyne laser from Gazebo. The original code changes the goal position and the start position randomly at every iteration. Because the goal is to compare the algorithm with ANS and OccAnt with the same goal and start positions, we have modified the code in order to load these positions from the same json file as the Habitat algorithms. Distance to the goal in the original code is calculated in a Euclidean way, but we have added also the possibility to calculate the obstacle-free distance, using the related function provided by the Habitat simulator.

4.2.2. Frontier exploration in ROS-X-Habitat

Frontier exploration for ROS is implemented with three packages:

- **Gmapping**: this package is used to provide laser-based SLAM [12]. Gmapping package creates a 2D occupancy grid map of the environment using laser readings and localizes the robot in the constructed map.
- **move_base**: The move_base package provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and a local planner to accomplish its global navigation task [21].
  - Global planner: it keeps a global map of the environment and tries to create a path to the goal from the current agent position.
  - Local planner: it keeps a local map of the environment currently surrounding the agent. This planner uses the local map and the path provided by the global planner in order to output the movement commands needed to execute the path.
- **explore_lite**: this package provides greedy frontier-based exploration. It takes as input the map produced by Gmapping and it uses move_base to reach the frontier selected at each step.

In order to implement the ROS frontier exploration algorithm in Habitat simulator we have used ROS_X_Habitat with the architecture represented in Figure 4.6.

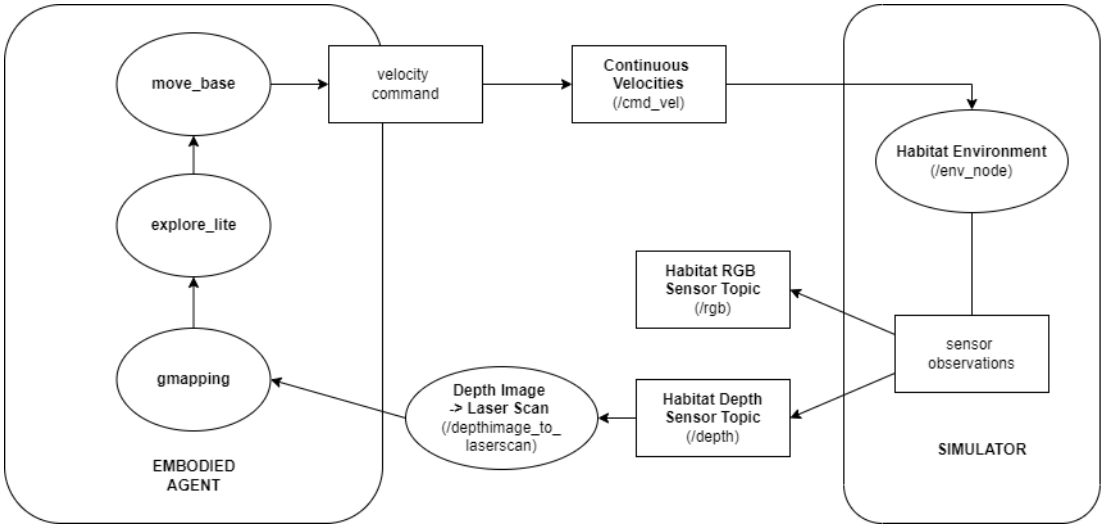


Figure 4.6: Schema of frontier exploration implementation with ROS-X-Habitat [29].

The node in charge of doing SLAM, Gmapping, receives as input the laser readings (from depthimage_to_laserscan node) and updates the map of the environment. The updated map is used by explore_lite node to perform frontier exploration and select the next frontier to visit, provided to move_base. In turn, move_base uses Dijkstra algorithm to plan the path to the next frontier and, using Trajectory Rollout algorithm [31], it publishes the velocities that are received by the Habitat Environment node.
The explore_lite node stops exploration when no more unexplored frontiers are present.

Frontier exploration in Stage simulator
Because of the problems related to incorrect laser readings and invisible textures with frontier exploration in the Habitat simulator highlighted in Section 3.5 we have also tested this algorithm in a 2D environment using Stage simulator. Maps are 2D *.png obtained from the Gibson dataset *.obj files through Utilities for Gibson Environment (Section 4.1.6). Frontier exploration, as in the 3D case, is obtained with the use of Gmapping, explore_lite, and move_base packages.

4.2.3. ANS and OccAnt

ANS and OccAnt are tested with the original code provided by authors in [22]. The simulation directly runs in the Habitat simulator. For both OccAnt and ANS we have used the pretrained models loaded on [22].

5 | Experimental results

In this chapter, we describe the experiments performed to compare the different algorithms (OccAnt, ANS, DRL, and frontier exploration) on exploration for map building and point-goal driven exploration tasks.
In Section 5.1, we describe all the elements used to compare the algorithms, specifying the ones used for exploration for map building and the ones used for point-goal driven exploration.
In Section 5.2, we describe the procedure used to compare the different aspects of the proposed algorithms.
In Section 5.3, we describe all the tests done and the results of the comparison between the algorithms in exploration for map building task.
In Section 5.4, we describe all the tests done and the results of the comparison between the algorithms in point-goal driven exploration task.

5.1. Comparing different methods

In this section, we describe the datasets, environments, and metrics used in order to compare the different exploration algorithms. The metrics in exploration for map building and point-goal driven exploration evaluate different elements.

5.1.1. Gibson dataset environments

Both exploration for map building and point-goal driven exploration experiments are performed on the Gibson dataset [10] in Habitat simulator. Gisbon dataset is a reproduction of real indoor spaces (made with 3D scanning and reconstructions).
In the comparison tests, we have considered 14 environments from the dataset, presented in Table 5.1. All the environments selected, that are shown in Figure 5.1 and Figure 5.2 are not used during the training of OccAnt and ANS. In the figures are also represented examples of paths followed during point-goal driven exploration episodes: in green the starting position and in red the positions crossed to reach the goal.

	area		area		area		area
Pablo	37.789	**Sisters**	94.529	**Elmira**	42.795	**Eudora**	37.079
Cantwel	107.582	**Ribera**	50.645	**Denmark**	40.796	**Swormville**	66.793
Scioto	512.963	**Greigsville**	43.603	**Sands**	153.132	**Eastville**	121.44
Edgemere	23.62	**Mosquito**	419.697				

Table 5.1: Area (m^2) of Gibson environments used in comparison.

We divided the environments in three groups by considering their area.

- **Small area:** 0 m^2 <= area < 70 m^2

  Edgemere, Eudora, Pablo, Denmark, Elmira, Greigsville, Ribera, Swormville
- **Medium area:** 70 m^2 <= area < 140 m^2

  Sisters, Cantwell, Eastville
- **Big area:** area >= 140 m^2

  Sands, Mosquito, Scioto

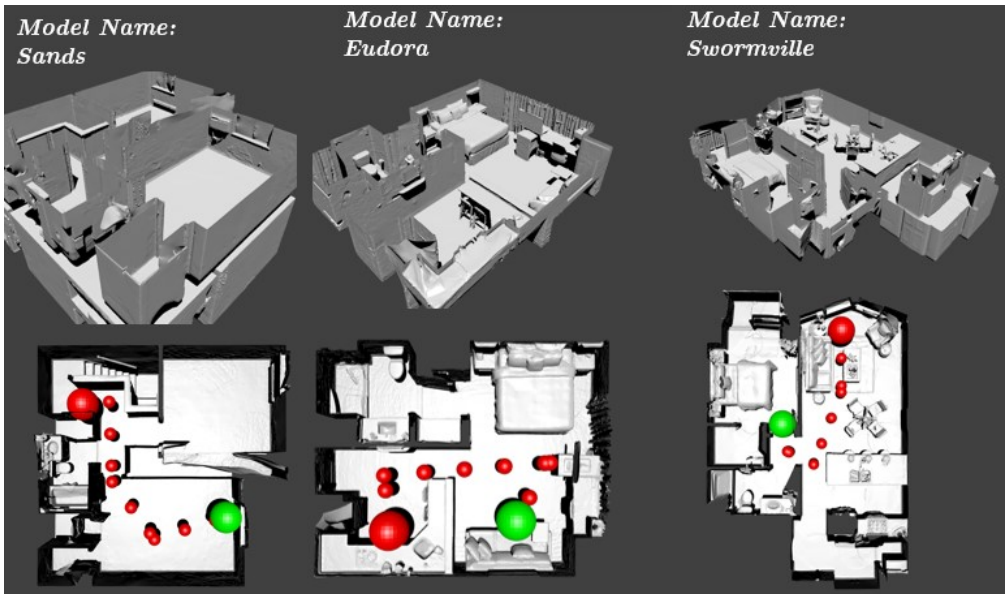


Figure 5.1: Gibson dataset environments - part 1.

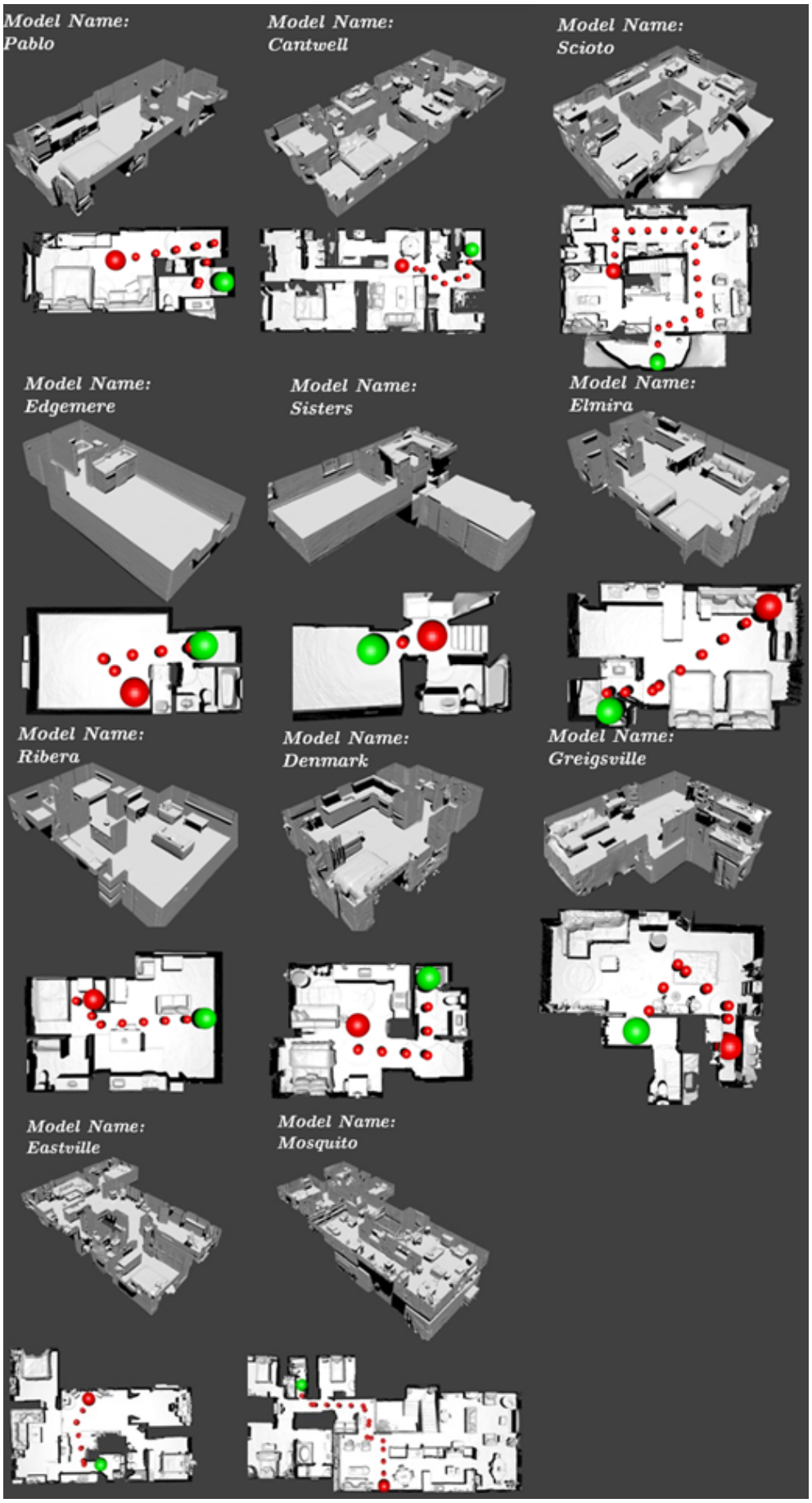


Figure 5.2: Gibson dataset environments - part 2.

5.1.2. DRL environment

The DRL algorithm is trained on smaller environments in the Gazebo simulator with a small number of furniture elements and only one room and consequently no corridors. All the DRL environments have the same size and at every algorithm interaction, some random boxes are placed in the room. Textures are not photorealistic and pretty uniform and all the elements in the room are taller than the exploring robot. A Deep Learning agent trained in this simple environment may not perform well when tested in more complex and realistic environments. In this kind of environment, represented in Figure 5.3, the robot has to avoid only few obstacles in order to reach the point-goal.

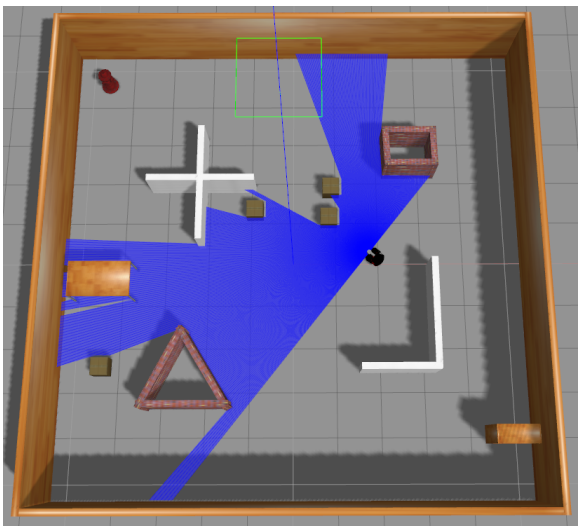


Figure 5.3: Example of DRL environment.

5.1.3. Metrics

In this section, we describe the metrics used in order to compare the algorithms in the two exploration tasks taken into consideration: exploration for map building and point-goal driven exploration. Some metrics are directly provided with algorithms implementation, others have been added by us.

5.1.3.1. Exploration for map building metrics

- **map_accuracy** (m^2)**:** the area of the global map built during exploration that matches the ground-truth provided. Directly provided by OccAnt and ANS.
- **area_seen** (m^2)**:** the total area seen during exploration. Directly provided by OccAnt and ANS.

- **free_space_seen** (m^2)**:** the total free space seen during exploration. Directly provided by OccAnt and ANS, added in frontier exploration.
- **occupied_space_seen** (m^2)**:** the total occupied space seen during exploration. Directly provided by OccAnt and ANS.
- **area_seen_over_time** (m^2/s)**:** the ratio between area_seen and episode time. Directly provided by OccAnt and ANS.
- **AC/AS:** the ratio between map_accuracy and area_seen. The more the value is close to one, the more the area is mapped accurately. Added in thesis comparison.
- **path_length** (m): the total length of the path followed during the exploration episode. We have calculated the path_length as the sum of the Euclidean distances between each step. This metric allows us to understand how much the agent has to explore the environment in order to be able to correctly map it. Added in thesis comparison.

5.1.3.2. Point-goal driven exploration metrics

- **success_rate:** the ratio between the total number of successes (robot distance to goal is lower than 0.3 m) and the number of comparison episodes. Directly provided by OccAnt and ANS, added in DRL.
- **SPL:** the ratio between the minimum obstacle-free path distance to the goal and the maximum value between the minimum obstacle-free path distance to the goal and the current distance travelled by the robot during the episode. This value is multiplied by 1 if the episode is successful, otherwise by 0. The closest the value is to 1, the closest the actual path to goal is to the optimal path.
  Directly provided by OccAnt and ANS, added in DRL.

```
SPL = success * (start_end_episode_distance /
max(start_end_episode_distance,agent_episode_distance))
```

- **SoftSPL:** is calculated similarly to SPL, but instead of multiplying to 0 or 1, it multiplies to the max values between 0 and 1 - the ratio between distance to target and the minimum obstacle-free path distance to target. In this way it is not 0 if the episode is not successful, but it is a measure of how much the robot is following the right path to the object. If the episode is successful the value is very close to SPL.

```
ep_soft_succ = max(0, 1 - distance_to_goal /
                              start_end_episode_distance)

SoftSPL = ep_soft_succ * (
    start_end_episode_distance /
    max(start_end_episode_distance, agent_episode_distance)
)
```

Directly provided by OccAnt and ANS, added in DRL.

- **dist_to_goal:** the distance to the goal (in m) at the end of the episode. The lower the value, the closest every episode ends to the goal. Directly provided by OccAnt and ANS, added in DRL.
- **num_steps:** the total number of steps taken during the episode. Directly provided by OccAnt and ANS, added in DRL.
- **avg_reward:** the average of the rewards obtained during the episode. Big values mean a high ability to reach the goal. Only provided in DRL.
- **avg_col:** the average of the collisions value obtained during the episode. Small values mean a high ability to avoid obstacles. Only provided in DRL.
- **path_length:** the total length of the path (in m) followed during the episode.

SPL and SoftSPL are computed using the obstacle-free path distance, but the original DRL code uses the Euclidean distance. As described in Chapter 4, we have added also the possibility to use the obstacle-free path distance in the code. Because the original implementation of the code doesn't use the obstacle-free path distance we have done experiments with both the Euclidean and the obstacle-free path distance.
SPL and SoftSPL as metrics are used only in episodes with obstacle-free path distance.

5.2. Comparison procedure

In this section, we describe how the different algorithms have been compared.

	DRL [60]	**ANS** [48]	**OccAnt** [97]	**Classical** [8]
DRL environment	V	X	X	X
Gibson environment	V	V	V	V
Exploration for map building task	X	V	V	V
map_accuracy	X	V	V	X
area_seen	X	V	V	X
free_space_seen	X	V	V	V
occupied_space_seen	X	V	V	X
area_seen_over_time	X	V	V	X
AC/AS	X	V	V	X
path_length	X	V	V	V
Point-goal driven exploration task	V	V	V	V
success_rate	V	V	V	V
SPL	V	V	V	X
SoftSPL	V	V	V	X
dist_to_goal	V	V	V	X
num_steps	V	V	V	X
avg_reward	V	X	X	X
avg_col	V	X	X	X
path_length	X	X	X	V

Table 5.2: Summary of the environments, tasks and metrics taken into consideration in order to test the different algorithms.

Table 5.2 shows the different environments where the algorithms are tested, the different tasks they try to accomplish, and the metrics used in order to compare them.
Because of the high GPU power required to train ANS and OccAnt we were not able to train the models, but we have used the pretrained models loaded on [22]. ANS and OccAnt are indeed trained with DD-PPO [118] method for distributed Reinforcement Learning in resource-intensive simulated environments with 8 GPUs and 16/32GB memory per GPU [6].
ANS or OccAnt can use RGB or depth as input and so the tested versions are:

- **ANS (depth):** ANS with depth input.
- **ANS (rgb):** ANS with RGB input.

- **OccAnt (depth):** OccAnt with depth input.
- **OccAnt (rgb):** OccAnt with RGB input.
- **OccAnt (rgbd):** OccAnt with RGB and depth input.

OccAnt and ANS are compared also in noise free and noisy simulations.
The pretrained ANS models loaded on [22] don't consider rgbd as input, but only rgb and depth inputs.
DRL, as also shown in Section 4.2.1, only uses depth inputs from the laser readings. RGB images are provided only as user visual support by ROS-X-Habitat simulator environment.
Frontier exploration only uses depth inputs.

Comparison platform

The computer on which we have tested the algorithm is equipped with an Nvidia GeForce GTX 970 GPU with 4GB of dedicated graphics memory. This of course influences the results, in particular the parts concerning time_per_episode metric. Despite the GPU is not one of the fastest on the market, the results are still useful because they provide a comparison of the execution time of the different algorithms on the same platform (provided in Section 5.3).

Examples of maps

In Figure 5.4 we show the kind of maps produced by OccAnt and ANS (on the top) and frontier exploration (on the bottom):

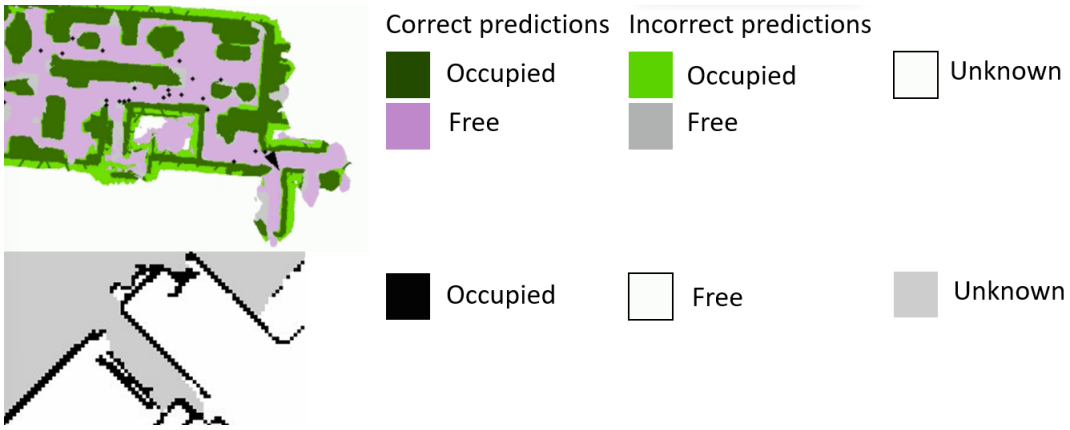


Figure 5.4: Example of map produced by OccAnt and ANS (on the top), example of map produced by frontier exploration (on the bottom).

5.3. Exploration for map building results

In this section, we present the results of the comparison done on exploration for map building task.
In Section 5.3.1 and Section 5.3.2, we compare the results between OccAnt and ANS in 3D environments.
In Section 5.3.3, we compare results obtained by OccAnt, ANS, and frontier exploration using both 2D and 3D environments. In Section 5.3.4, we show a comparison of OccAnt, ANS, and frontier exploration on the decision-making times.

5.3.1. OccAnt vs. ANS exploration for map building results - Noise free

The results here are obtained in the Habitat simulator in a setting where noise is not taken into consideration. In this comparison, we have used the pretrained models loaded on [22]. The weights of the pretrained models are obtained from the algorithms trained in noise free environments. We have decided to compare OccAnt and ANS alone before comparing them to frontier exploration because they run directly in the Habitat simulator. Every exploration episode that runs directly in the Habitat simulator is very fast and so we have been able to test these two algorithms on a bigger number of episodes and environments. In every one of the 14 maps from the Gibson dataset described in Section 5.1.1 we have run 71 episodes (each one from a different starting location). The 71 episodes are directly provided by the Gibson dataset for testing. The maximum number of steps (actions taken by the agent) is 500 and after that exploration is stopped.
OccAnt (depth) and OccAnt (rgbd) with the pretrained models loaded on [22] perform very poorly when compared to OccAnt (rgb) and don't produce as output map_accuracy metric, highlighting some problems in the pretrained models. As already said, due to the high computational power required, we can't retrain these two algorithm implementations on our platform and so we have not considered them in this section (while they have been considered in point-goal driven exploration task in Section 5.4.1).
Compared to the metrics originally considered in ANS and OccAnt we have also added area_seen_over_time metric because considering only the area_seen metric in the episode may not lead to a fair comparison. An agent, for example, could explore more areas, but in a long time and so area_seen metric alone doesn't provide a piece of very useful information.

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	47.451 (17.449)	43.478 (15.324)	**48.725 (18.142)**
area_seen	55.609 (19.849)	53.841 (18.872)	**56.913 (21.031)**
free_space_seen	29.801 (11.002)	29.059 (10.743)	**30.991 (12.010)**
occupied_space_seen	25.808 (9.437)	24.782 (8.794)	**25.922 (9.415)**
time_per_episode	**0.865 (0.002)**	0.908 (0.002)	0.984 (0.004)
area_seen_over_time	**64.288 (22.242)**	59.296 (20.382)	57.838 (20.072)
AC/AS	0.853 (0.045)	0.808 (0.048)	**0.856 (0.042)**

Table 5.3: Results of exploration for map building done in all the environments with OccAnt and ANS in noise free simulation.

Table 5.3 presents the results of the complete comparison of 994 episodes on the 14 environments. In parentheses, we report also the standard deviation. Results breakdown on small, medium, and big environments are shown in Appendix A.1.
OccAnt (rgb) proved to be the best choice for different metrics (map_accuracy, area_seen, free_space_seen, and occupied_space_seen), except in small environments, where it results to be really similar to ANS (depth).
As already highlighted, time_per_episode metric strongly depends on the GPU power available to run the simulation episodes. Despite this limitation, the time_per_episode is a useful metric, as it shows us how OccAnt (rgb) obtains better map_accuracy, area_seen, free_space_seen, and occupied_space_seen at the expense of a longer exploration time with respect to ANS (depth).
Despite OccAnt (rgb) being trained in order to anticipate the occupancy, there is no big difference in occupied_space_seen metric.
ANS (depth) area_seen metric is lower than OccAnt (rgb), especially in big environments, but the ratio AC/AS (map_accuracy over area_seen) is in all cases comparable, proving that ANS (depth) explores less (as we said few lines above it also has the lowest in time_per_episode metric), but it keeps the same level of map_accuracy over area_seen. The introduction of time_per_episode (not originally considered in the authors' works) allows us to better understand the performance of the two algorithms: despite area_seen value at the end of the 500 steps in ANS (depth) being lower than the value obtained by OccAnt (rgb), ANS (depth) is able to explore more area in the same time. In any case, with higher average values come also higher standard deviation values.
These results strengthen the importance of selecting metrics really able to identify the strengths or weaknesses of every algorithm, especially when we compare Deep Learning

algorithms and classical algorithms.
The presence of occupancy anticipation module in OccAnt (rgb) doesn't give to OccAnt (rgb) a clear advantage in exploration for map building task, but in point-goal driven exploration task (as later seen in Section 5.4.1) it allows better performance.

5.3.2. OccAnt vs. ANS exploration for map building results - Noisy

The following results are obtained in Habitat simulator and noise is taken into consideration. As already said, in this comparison, we have used the pretrained models loaded on [22]. The weights of the pretrained models are obtained from the algorithms trained in noisy environments. Even if, as highlighted in Section 2.5.1, the noise model may not be a perfect reconstruction of the one we have in the real world, the results obtained here are useful to understand how algorithms' performance changes with different world configurations. The maximum number of steps (actions taken by the agent) is 500 and after that exploration is stopped.

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	40.900 (13.426)	37.958 (11.391)	**41.218 (13.740)**
area_seen	51.816 (16.321)	50.574 (15.138)	**54.062 (18.122)**
free_space_seen	28.060 (10.298)	27.490 (9.736)	**29.599 (11.320)**
occupied_space_seen	23.757 (7.896)	23.085 (7.375)	**24.462 (8.548)**
time_per_episode	**0.843 (0.002)**	0.877 (0.002)	0.964 (0.005)
area_seen_over_time	**61.466 (19.339)**	57.677 (17.236)	56.081 (18.765)
AC/AS	**0.789 (0.056)**	0.750 (0.059)	0.762 (0.068)

Table 5.4: Results of exploration for map building done in all the environments with OccAnt and ANS in noisy simulation.

Table 5.4 shows the results of the complete comparison on 994 episodes on the 14 environments. In parentheses, we report also the standard deviation. Results on small, medium, and big environments are shown in Appendix A.2.
The results of this comparison done in noisy environments are comparable to the ones we have obtained in noise free environments and discussed in Section 5.3.1.
Even in this configuration OccAnt (rgb) proved to be the best algorithm concerning map_accuracy, area_seen, free_space_seen, and occupied_space_seen values while ANS (depth) obtains the best values for time_per_episode, area_seen_over_time, and AC/AS.

Also in noisy environments OccAnt (rgb) obtains very similar values with respect to ANS (depth) in map_accuracy, area_seen, free_space_seen, and occupied_space_seen metrics.
These results proved the ability of the two algorithms to obtain generalization in different kinds of environments and configurations. However, as already said in Section 2.5.1, the noise model on which the algorithms are trained and tested may not be so good which is why the algorithms should be also tested in the real world. In any case, with higher average values come also higher standard deviation values.
We can get more interesting considerations by comparing the results obtained in noise free and noisy environments. Results in noisy environments are expected to be lower compared to the ones obtained in noise free environments where movements can follow a deterministic trajectory.

	ANS (depth) noise free	**ANS (depth) noisy**	**ANS (rgb) noise free**	**ANS (rgb) noisy**	**OccAnt (rgb) noise free**	**OccAnt (rgb) noisy**
map_accuracy	47.451 (17.449)	40.900 (13.426)	43.478 (15.324)	37.958 (11.391)	48.725 (18.142)	41.218 (13.740)
area_seen	55.609 (19.849)	51.816 (16.321)	53.841 (18.872)	50.574 (15.138)	56.913 (21.031)	54.062 (18.122)
free_space_seen	29.801 (11.002)	28.060 (10.298)	29.059 (10.743)	27.490 (9.736)	30.991 (12.010)	29.599 (11.320)
occupied_space_seen	25.808 (9.437)	23.757 (7.896)	24.782 (8.794)	23.085 (7.375)	25.992 (9.415)	24.462 (8.548)
area_seen_over_time	64.288 (22.442)	61.466 (19.339)	59.296 (20.382)	57.677 (17.236)	57.838 (20.672)	56.081 (18.765)
AC/AS	0.854 (0.045)	0.789 (0.056)	0.808 (0.048)	0.750 (0.059)	0.856 (0.042)	0.762 (0.068)

Table 5.5: Results of the comparison of exploration for map building done with OccAnt and ANS in noise free and noisy simulation.

Table 5.5 shows the differences between the executions of the algorithms in noise free and noisy conditions on the total 994 episodes in the 14 environments. In parentheses, we report also the standard deviation. All the metrics in noise conditions, as expected, present on the average lower values. All the algorithms have lower values in area_seen_over_time because, as expected, in noisy environments agents take more time in order to visit the same amount of area.
All the algorithms see less area (area_seen is reduced) and even the map_accuracy is reduced. In all the examined cases, the reduced AC/AS metric indicates a lower ability to correctly map the environments, even if less area is observed during the exploration episode. In both cases, however, with higher average values come also higher standard deviation values.

It is also useful to consider the ratio between AC/AS in noisy and noise free environments for the same algorithm, $\frac{\text{AC/AS}_{\text{noisy}}}{\text{AC/AS}_{\text{noisefree}}}$:

- **ANS (depth):** 0.924
- **ANS (rgb):** 0.928
- **OccAnt (rgb):** 0.890

Observing these values it is clear that OccAnt (rgb) is the one that suffers the major reduction for the AC/AS metric. However, even if AC/AS is reduced in the noisy environments, it still shows a good value for all the three algorithms.
From what we have done in this section it becomes apparent that if we use different simulation settings (noisy vs. noise free environments) results can change significantly. Because of this, these algorithms should also be tested in the real world in order to understand if their behavior remain similar to the one obtained in the simulator or if they only have learnt how to exploit specific elements of the simulator.

Figure 5.5: Examples of map produced in ANS (depth), ANS (rgb), and OccAnt (rgb) in noise free and noisy simulation.

In Figure 5.5 we can see examples of maps produced by the three different algorithms in noise free and noisy environments. Because the path chosen between the list of the possible paths in noisy environments (Figure 2.21), sometimes the area_seen is sensibly reduced as shown in figure by ANS (rgb) algorithm. However, as shown in the tables, the average value on 994 episodes is not so different from the one obtained in noise free environments.

Results in environments of different size

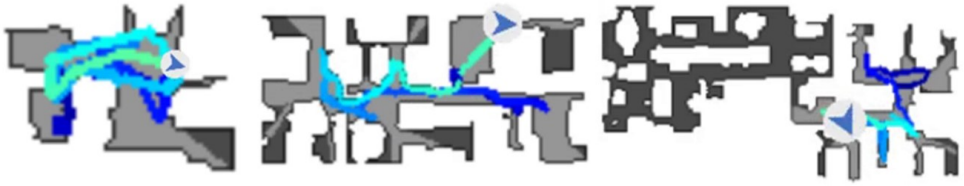


Figure 5.6: Exploration path by OccAnt(rgb) in small, medium, big environments in exploration for map building.

Figure 5.6 shows the path (light blue line) followed during exploration for map building by OccAnt (rgb) in small, medium, and big environments. In the figure dark grey indicates the area not seen, while light grey indicates the area seen during the episode. The agent during exploration (in both noisy and noise free environments) often comes back to already visited places because it wants to achieve high accuracy. This behavior doesn't have an impact on the amount of area_seen in small and medium environments, but, as clearly visible in the figure, in big environments the agent doesn't get very far from the starting point. With this attitude, the agent gains high accuracy in the small area explored at the expense of exploring bigger areas.
Also ANS (rgb) and ANS (depth) often come back to already visited places during exploration episodes.
This behavior will be particularly considered in the next section, where ANS and OccAnt are compared to frontier exploration algorithm, which is characterized by an exploration path that comes back to already visited places with reduced frequency.

5.3.3. Frontier exploration vs. OccAnt vs. ANS

In this section, we show the results of the comparison done between frontier exploration (both in 2D, Stage simulator, and 3D, Habitat simulator), OccAnt, and ANS (both in Habitat simulator).
In these tests, we have used path_length as a metric in order to compare the three different implementations of exploration algorithms because of the difficulties in using some of the metrics proposed by OccAnt and ANS with frontier exploration in 3D. Furthermore, in Section 5.3.1, we have shown that only considering area_seen metric may not provide a fair comparison of the different algorithms. Because of this, in this section, we analyze the area explored and mapped with a specific path_length (specified later).
In the following tests, all the algorithms use the same initial position and rotation in the environment. For performance reasons, we have tested the three algorithms only on some maps, but with significant differences in size.
Frontier exploration algorithm uses laser readings as input and sometimes these readings in the Habitat simulator with Gibson dataset don't fully represent a real world situation. As it can be seen in Figure 5.7 the textures present holes and so laser readings go beyond walls or objects creating frontiers that don't exist in the real world.

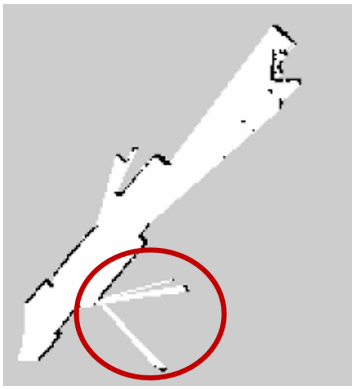


Figure 5.7: Example of incorrect laser readings with frontier exploration in Gibson environments.

Sometimes the agent gets stuck due to some inaccurate textures and it can't move anymore, so the frontier exploration algorithm stops its iterations because the agent can't reach the previously saved frontiers. In all the different maps that we have tested, the agent gets stuck in some textures at some time and because of this we have decided to stop exploration in ANS (rgb), ANS (depth), and OccAnt (rgb) when path_length is the same than the one obtained with frontier exploration, in order to get comparable results. In order to propose a more qualitative analysis of the frontier exploration algorithm and

discover what happens when the agent doesn't get stuck, we have used the tool Utilities for Gibson environments discussed in Section 4.1.6 in order to evaluate frontier exploration algorithm also in 2D environments. Using 2D floor maps with Stage simulator we don't have anymore the texture holes problems and the robot never gets stuck in broken textures. On the other hand, we can have different path_length because all the objects in the environments are flattened on the ground and different frontiers can be identified by the algorithm. The maps produced by OccAnt and ANS represent obstacles in a different way with respect to the one proposed by Gmapping SLAM and because of this, we have decided to compare only the free_space_seen.

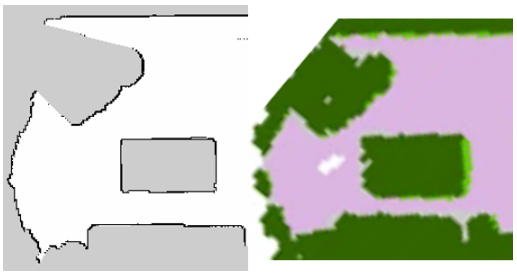


Figure 5.8: Obstacles representation (Gmapping on the left, ANS/OccAnt on the right).

As shown in Figure 5.8, we can see how Gmapping SLAM considers the obstacle area of the table only the black line on the outline, while ANS and OccAnt consider as obstacle area all the area painted in green. For this reason, occupied_space_seen would not be fair.
Free_space_seen with frontier exploration is proposed only in the 2D case because in 3D, as already said, there are a lot of holes in the textures and the obtained value would not represent a real situation.
In the following pages we consider exploration episodes in one environment at a time.

5.3.3.1. Greigsville

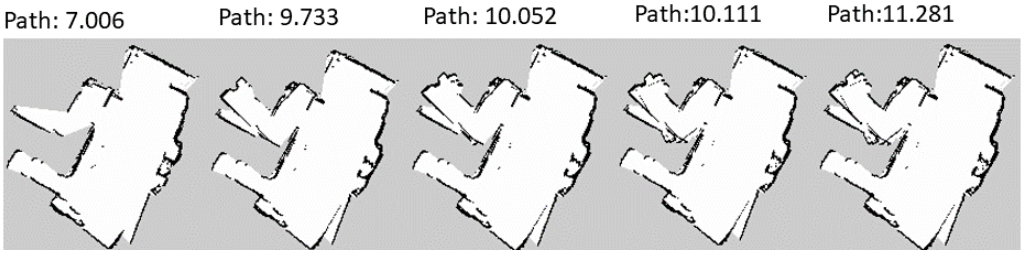
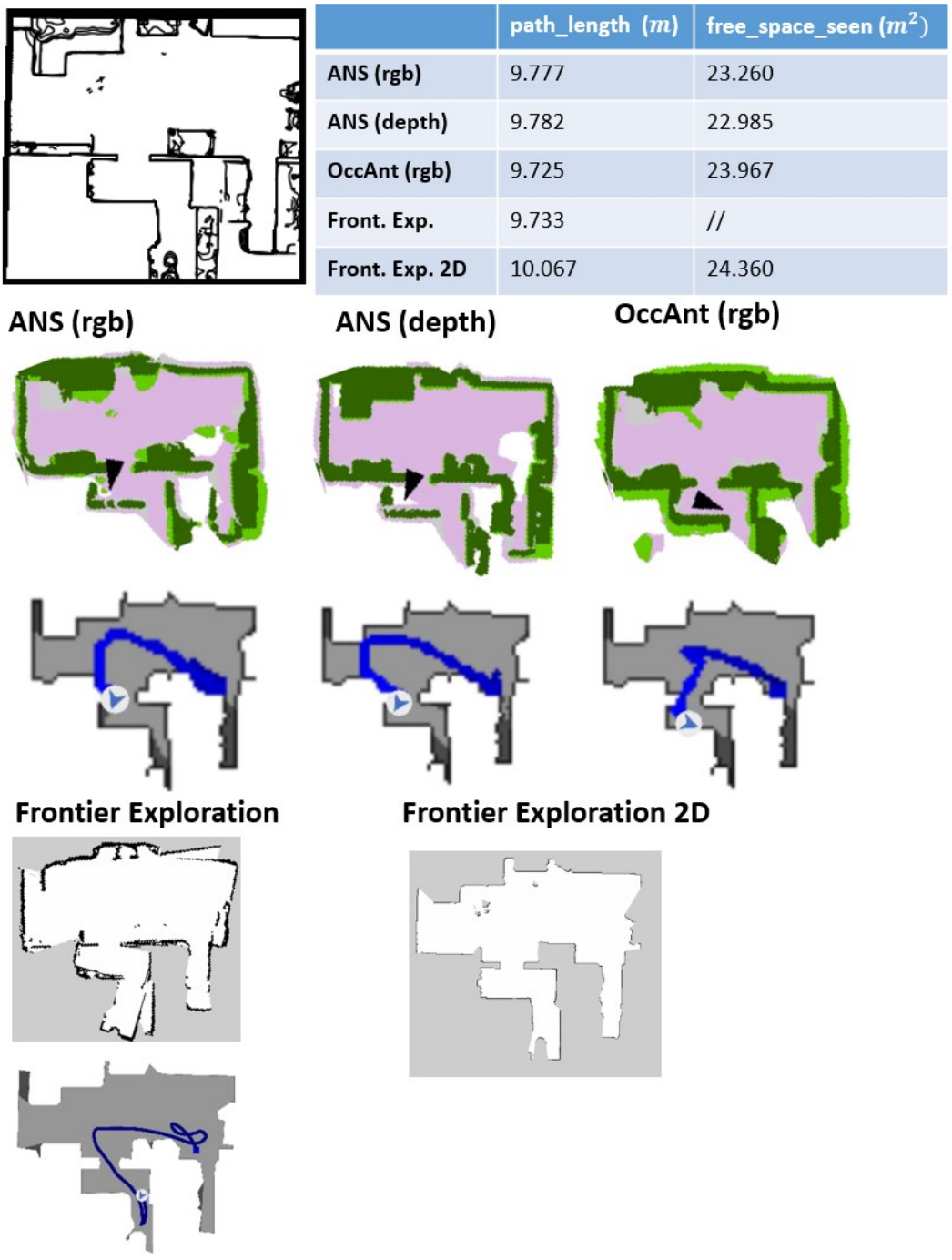


Figure 5.9: Example of map produced by frontier exploration in 3D Greigsville.

As discussed earlier, from Figure 5.9 we can see that agent with frontier exploration gets stuck in an obstacle after traveling 9.733 with frontier exploration in 3D environment. The quality of the map after 11.281 m is worse than after 9.733 m because the agent is stuck in the textures and it is rotating in place.
In Figure 5.10 we show the results of a single exploration episode stopped at 9.733 m.



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	9.777	23.260
ANS (depth)	9.782	22.985
OccAnt (rgb)	9.725	23.967
Front. Exp.	9.733	//
Front. Exp. 2D	10.067	24.360

Figure 5.10: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Greigsville.

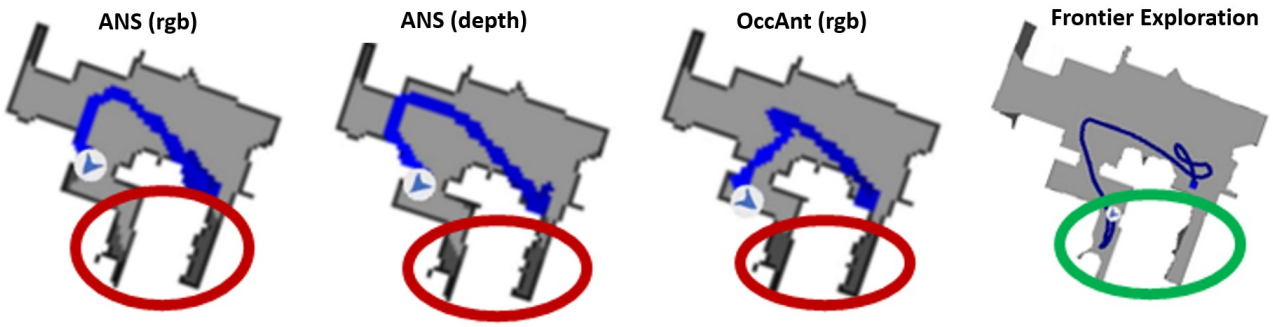


Figure 5.11: Path followed by ANS (rgb), ANS (depth), OccAnt (rgb), and frontier exploration in Greigsville.

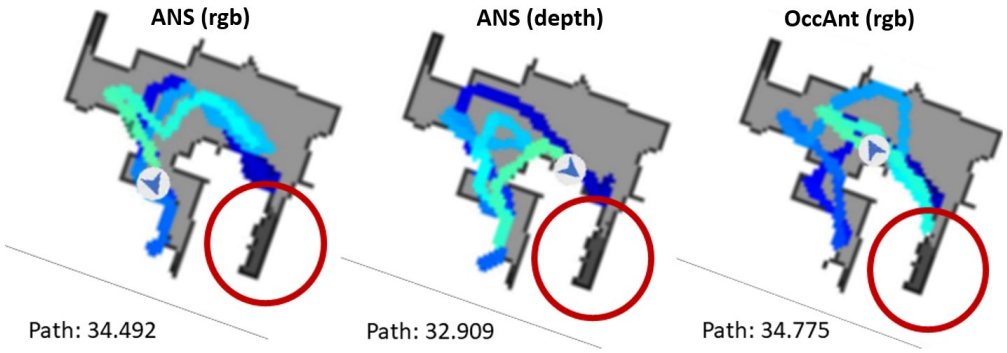
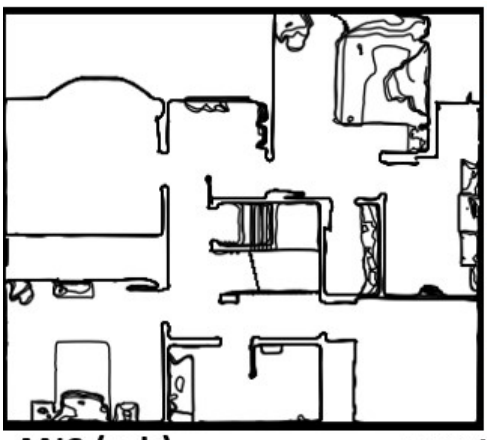


Figure 5.12: Path followed by ANS (rgb), ANS (depth), and OccAnt (rgb) in Greigsville with path length not limited to 9.733 m.

From Figure 5.10 we can see the different maps produced by the algorithms.
The agent with frontier exploration (in both 2D and 3D) is able to map more area. The agent with frontier exploration in 3D is able to explore more area, as shown in Figure 5.11 when compared to OccAnt and ANS.
The area in the right part of the red circle is correctly predicted by ANS and OccAnt algorithms, but even with a very high path_length (as verified in Figure 5.12) it is never visited.
The area in the left part of the red circle is visited by frontier exploration agent event with a low path_length, while it is visited by OccAnt and ANS only with an higher path_length, as shown in Figure 5.12.
This behavior may lead to a correct mapping of the perimeter of an area, without knowing the actual obstacles contained in it.
In Appendix A.3 we propose other two comparisons run in small environments. In both the situations agent with frontier exploration (3D) is able to map more area than the other algorithms with the same path_length.

5.3.3.2. Scioto



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	30.424	38.607
ANS (depth)	30.437	41.952
OccAnt (rgb)	30.444	45.783
Front. Exp.	30.448	//
Front. Exp. 2D	30.869	53.785

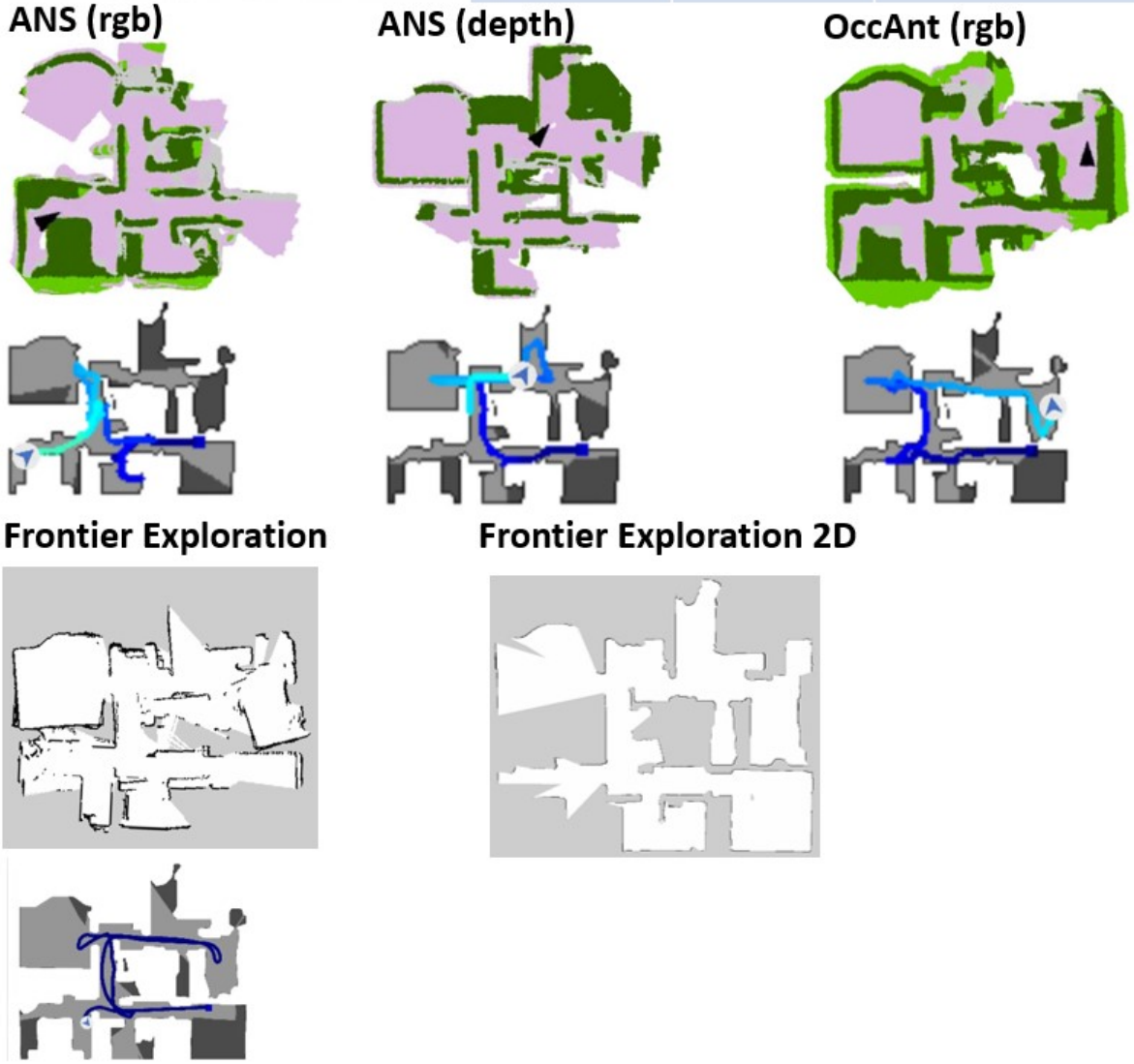


Figure 5.13: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Scioto.

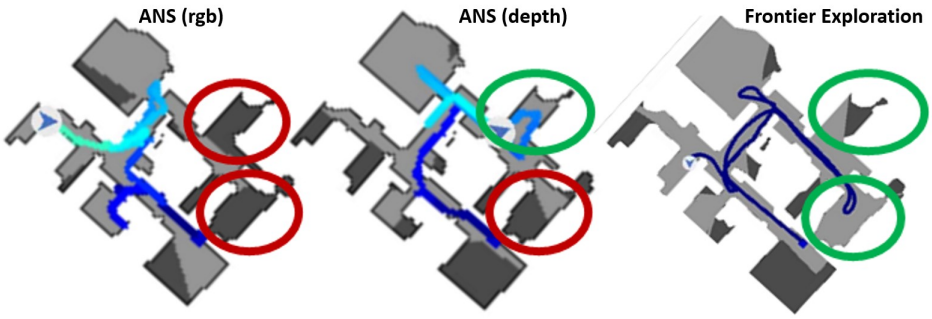


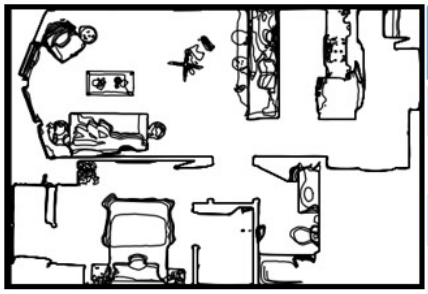
Figure 5.14: Path followed by ANS (rgb), ANS (depth), and frontier exploration in Scioto.

Figure 5.15: Example of incorrect laser readings in Scioto.

Also in this exploration episode frontier exploration algorithm agent in 3D gets stuck in an obstacle and can't reach the other frontiers available. However, it still gets comparable results when we consider path_length metric. The results of the single exploration episode stopped at 30.448 m are shown in Figure 5.13. Looking at the area explored in Figure 5.14 we can compare frontier exploration in 3D with ANS (rgb) and ANS (depth). Frontier exploration 3D agent sees two rooms that ANS (rgb) agent doesn't reach because its path is concentrated on a smaller area of the floorplan (it doesn't explore the right part of the floorplan) with the same path_length. Also ANS (depth) path is concentrated on a smaller area of the floorplan. This behavior, previously identified in Section 5.3.2, is particularly evident in this map: looking at the table in Figure 5.13, we can see that free_space_seen in ANS and OccAnt is sensibly lower than the one obtained by frontier exploration with the same path_length. OccAnt (rgb) is trained to achieve high mapping accuracy and so its agent concentrates its paths on a smaller area with respect to the path proposed by frontier exploration algorithm. ANS (rgb) and ANS (depth) reward increases in area seen, but probably the reward function needs some tuning in order to explore more area. From Figure 5.15, in the red circle, we can see an example of incorrect readings obtained from laser only because of the holes in the texture in the simulator

environment. This behavior that worsens the mapping ability of frontier exploration in 3D of course will not happen in the real world (under reasonable assumptions on the sensors and types of walls) and the algorithm will obtain better results, as it is confirmed by frontier exploration 2D execution.

5.3.3.3. Swormville



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	38.414	30.903
ANS (depth)	38.640	34.340
OccAnt (rgb)	38.526	34.597
Front. Exp.	38.573	//
Front. Exp. 2D	38.308	31.354

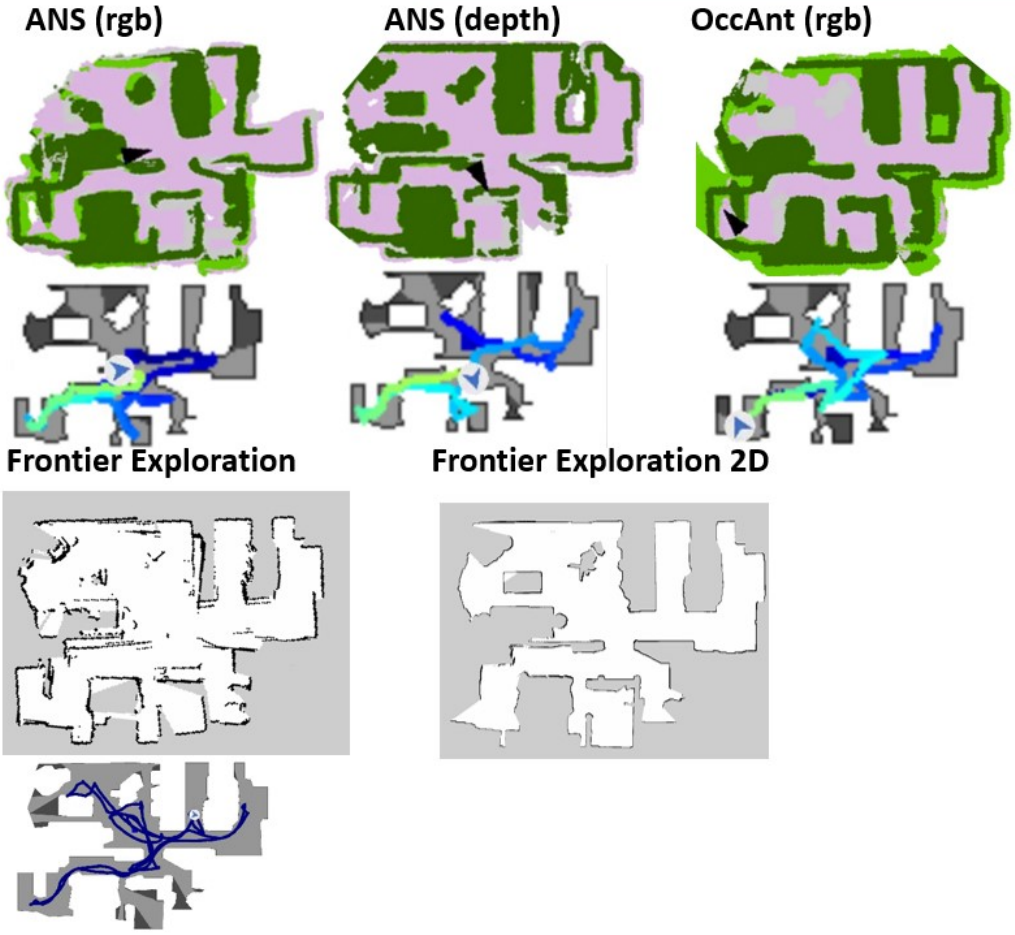
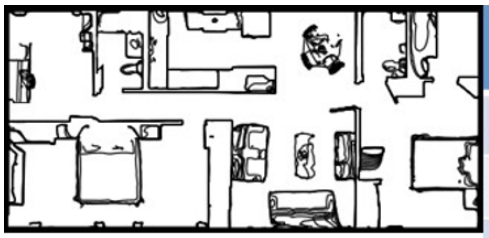


Figure 5.16: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Swormville.

In this comparison, frontier exploration 3D agent is able to explore more area without getting stuck in the textures for a longer time, and in fact, the algorithm, in this case, is able to map all the environment (even if there are the usual imperfections due to the presence of holes in the textures). Also in this environment, we can see how the path of frontier exploration agent covers more area with respect to the paths of ANS and OccAnt algorithms. In these two algorithms, the agent's behavior is to come back on the same path multiple times, as already highlighted in the previous comparison of Section 5.3.3.2. In this case, frontier exploration 2D, as it is shown from the map in Figure 5.16 doesn't visit all the rooms visited by frontier exploration in 3D probably because of the different path taken in 2D environment, but it is still comparable with OccAnt and ANS.

5.3.3.4. Cantwell



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	21.862	33.869
ANS (depth)	21.865	31.272
OccAnt (rgb)	21.844	41.637
Front. Exp.	21.816	//
Front. Exp. 2D	21.635	40.067

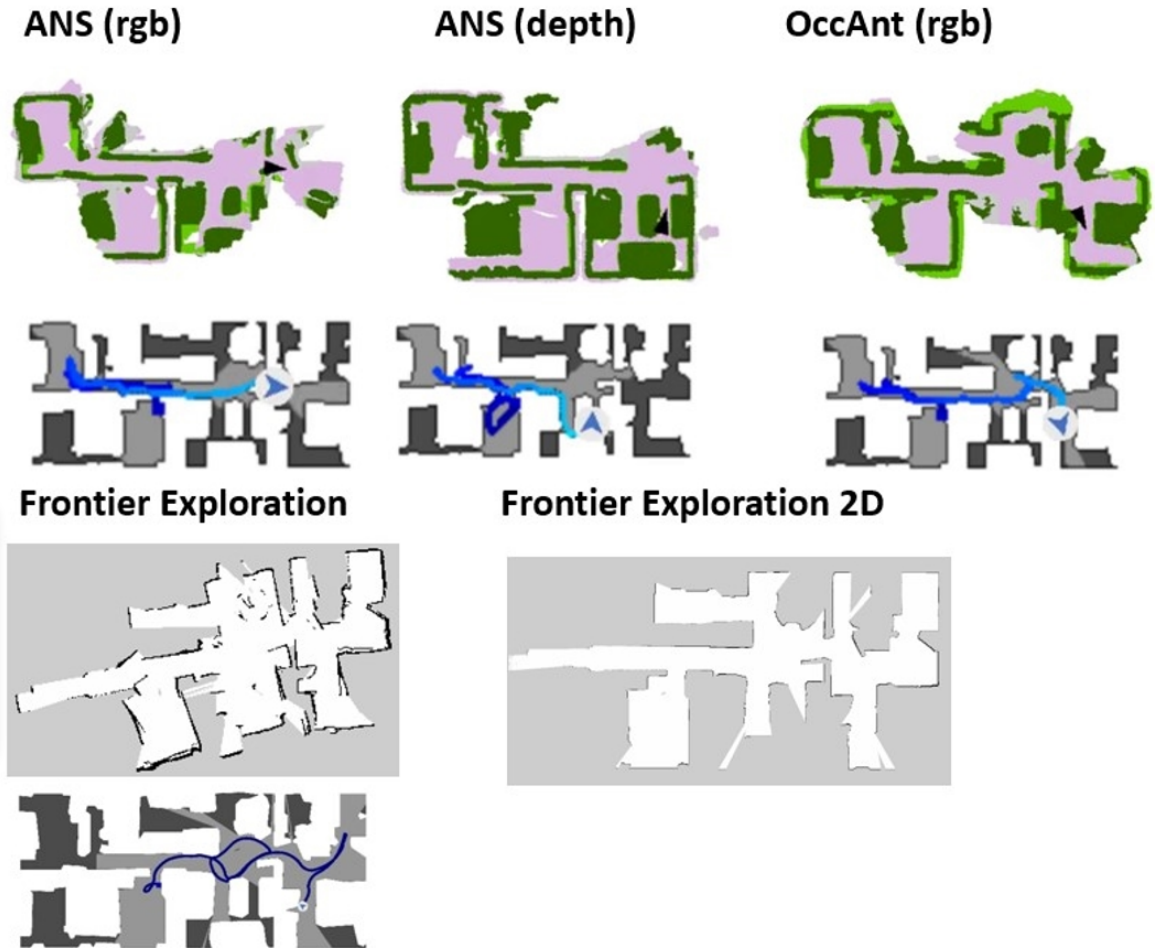


Figure 5.17: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Cantwell.

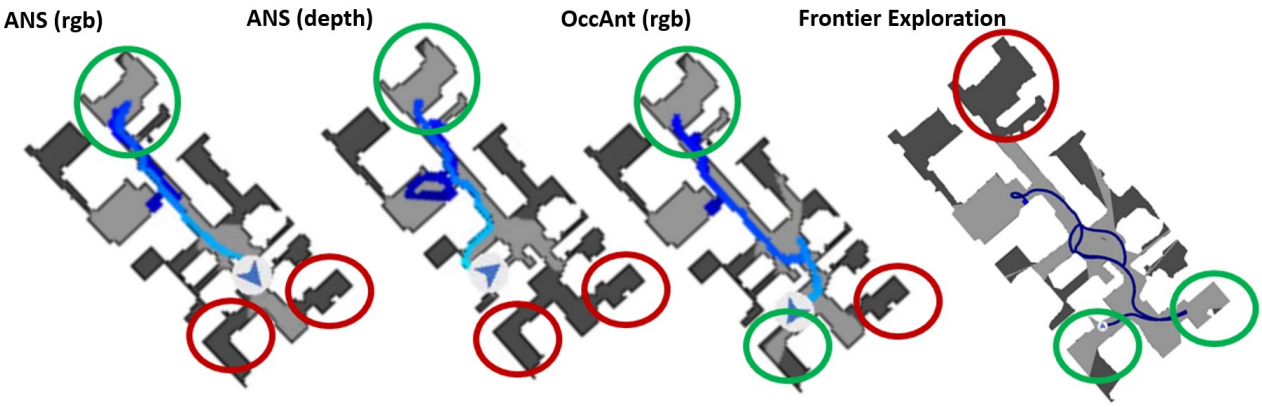
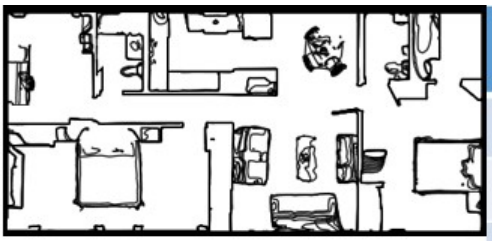


Figure 5.18: Path followed by ANS (rgb), ANS (depth), OccAnt (rgb) and frontier exploration in Cantwell.

During this exploration episode, frontier exploration 3D agent gets stuck two times, but after the second time it is not able to recover the right behavior and for this reason, we stop exploration after 27.491 m.
Looking at Figure 5.18, we can see that frontier exploration 3D explores one room more than ANS (rgb) and ANS (depth).
Looking at the table in the figure, we can see that frontier exploration 2D free_space_seen is comparable to the one obtained with OccAnt (rgb), which gets better results than ANS (rgb) and ANS (depth).



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	27.420	35.030
ANS (depth)	27.409	33.962
OccAnt (rgb)	27.413	36.210
Front. Exp.	27.491	//
Front. Exp. 2D	27.412	36.793

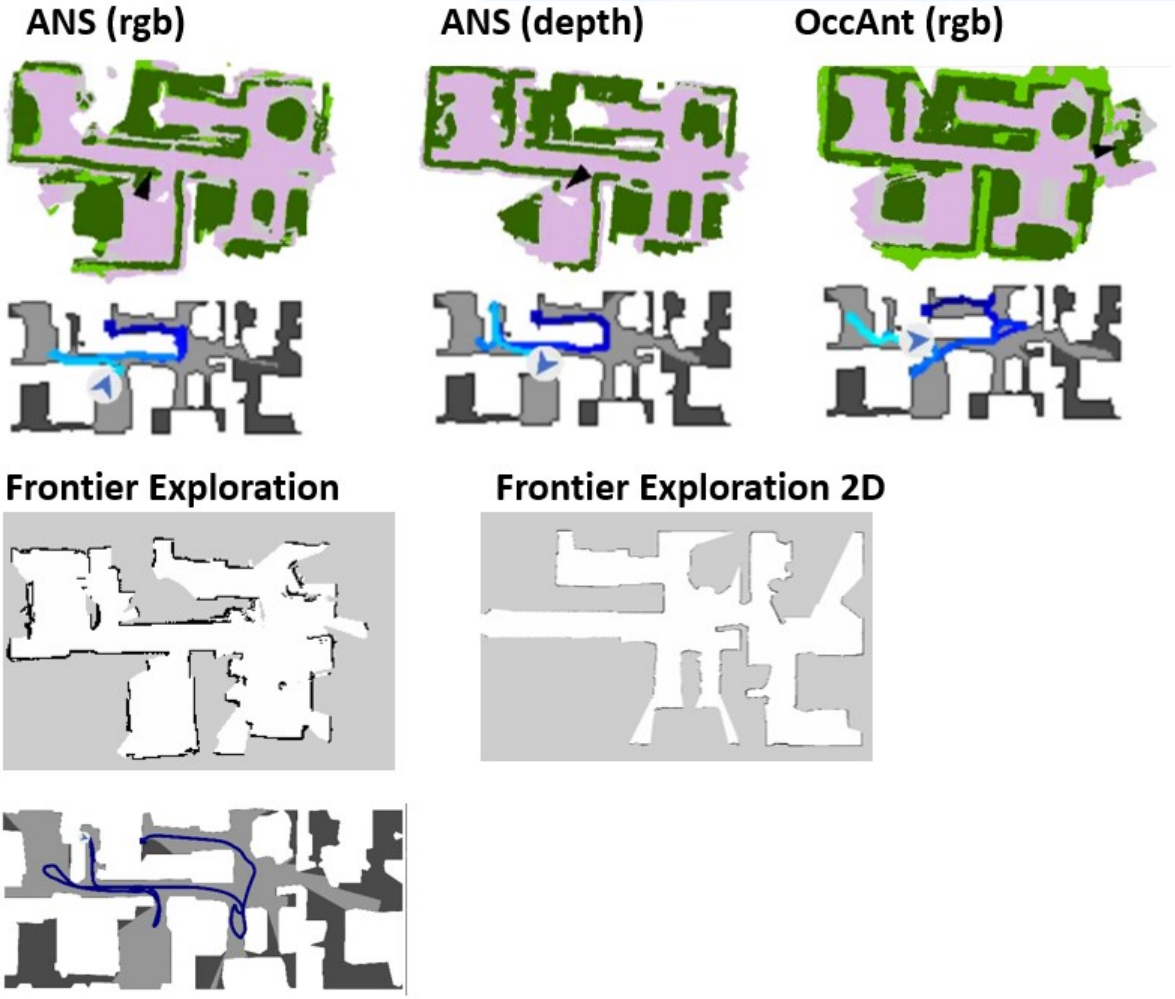


Figure 5.19: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Cantwell.

In Figure 5.19 we have conducted another comparison in the same map changing the starting point. In this episode, we obtain results comparable to the ones obtained in the previous run, proving that the results are consistent even if the starting point is different.

5.3.3.5. Results analysis

Frontier exploration (3D) agent finds some problems when deployed in the Habitat simulator with Gibson dataset because of the imperfections in the textures in the simulator environments. The holes in textures cause incorrect laser readings as described in Section 5.3.3. These incorrect laser readings are the cause of some inconsistencies in SLAM mapping, that are not present when we consider the 2D case with no holes in textures. Some other imperfections make the agent unable to resume navigation because it is stuck in some inaccurate textures. These are situations that most likely will not happen in a real world environment.

Considering the path_length of the exploration algorithms we can see how maps produced by frontier exploration (in 3D) are comparable with OccAnt and ANS. In frontier exploration 2D where no problems about textures are present, we can see that results are consistent with respect to ANS or OccAnt (concerning free_space_seen). The path followed in frontier exploration 2D sometimes is not the same as the one followed by the agent in 3D frontier exploration because of the fact that all the objects are flattened to the ground by the utility used to generate the map.

OccAnt and ANS most of the time have lower free_space_seen values because their exploration paths are concentrated on a smaller area than fronter exploration (2D and 3D) as can be clearly seen in big environments like those of Sections 5.3.3.2 and 5.3.3.3. This behavior comes from the fact that OccAnt (rgb) is trained to maximize mapping accuracy, while ANS (depth) and ANS (rgb) probably need some tuning of the reward function. Also in smaller environments like the ones shown in Sections 5.3.3.1 and A.3.1, even if less noticeable, they have this behavior, because, considering the same path_length, frontier exploration algorithm has higher value in free_space_seen metric.

In order to correctly compare frontier exploration, OccAnt, and ANS, it should also be considered that both OccAnt and ANS are trained with expensive techniques in this kind of environment, while frontier exploration can be simply deployed without any kind of particular optimization or expensive training.

Frontier exploration algorithm has already proven to provide comparable results when deployed in a real world scenario. OccAnt and ANS, on the other hand, as described in Section 5.3.2 can have different behaviors if the simulation settings change. Because of this reason, a future test in the real world should be done in order to verify that the results obtained in this section can also be obtained in real-world more complex environments.

After having shown that the selected classical and Deep Learning algorithms have comparable performance in exploration for map building task our goal is to understand also their behavior in the point-goal driven exploration task, in order to figure out if Deep

Learning techniques can provide an advantage in that task (Section 5.4.3).

5.3.4. Decision-making time comparison frontier exploration vs. OccAnt vs. ANS

In order to fully compare OccAnt, ANS, and frontier exploration it can be also useful to compare the decision-making time required by each algorithm to select the next action to be executed.
In OccAnt and ANS the simulator teleports the agent to the next location after the algorithm has produced the next action and the movement time can be approximate to zero. The time to select the next action to execute can so be approximate to the execution_time over num_steps.
In frontier exploration implementation, however, movement time can't be approximate to zero. Because of this, the decision-making time considered is the one required to find the frontiers, sort them, and select the next goal to follow. The time considered next is obtained from the average of the execution of the algorithm in the different environments. What comes out is that the decision-making time in OccAnt and ANS is lower than the one of frontier exploration, as can be seen here:

- **ANS (rgb):** 0.001807 s.
- **ANS (depth):** 0.001722 s.
- **OccAnt (rgb):** 0.001958 s.
- **Frontier exploration:** 0.095430 s.

Frontier exploration implementation doesn't use particular optimization techniques, while as already said in Section 4.1.4, OccAnt and ANS are specifically developed for Habitat simulator which focuses on execution time.
Frontier exploration decision-making time, however, is still low and fully compatible with use in the real world. OccAnt and ANS require a long training time and a lot of computational resources in order to be trained [6], while frontier exploration can be simply deployed in the environment.

5.4. Point-goal driven exploration results

In this section, we present the results of the comparisons on point-goal driven exploration. A point-goal is considered reached if the agent's distance to it is lower than 0.3 m. In Section 5.4.1 we propose a comparison of OccAnt and ANS in 14 environments. In Section 5.4.2 we present some tests done in order to understand how an algorithm (DRL) trained in simple environments works when deployed in more complex environments and the consequent difficulty in writing a reward function for Reinforcement Learning algorithm that can work well in different environments. In Section 5.4.3 we propose a comparison of OccAnt, ANS, and a classical algorithm in 3 different environments.

5.4.1. Point-goal driven exploration ANS vs. OccAnt results - Noise free

The results in this section are obtained from simulation episodes where no noise is present. As already said, in this comparison, we have used the pretrained models loaded from [22]. The weights of the pretrained models are obtained from the algorithms trained in noise free environments. Because of the fact that both OccAnt and ANS directly run in the Habitat simulator, we have been able to test these two algorithms on a big number of episodes and environments. In every one of the 14 maps from Gibson dataset described in Section 5.1.1, we have run 71 episodes (each one with a different starting location and goal location). The 71 episodes are directly provided by the Gibson dataset for testing. The maximum number of steps (actions taken by the agent) is 500 and after that point-goal driven exploration is stopped.

	ANS (depth)	ANS (rgb)	OccAnt (depth)	OccAnt (rgb)	OccAnt (rgbd)
succes_rate	0.722	0.316	**0.867**	0.808	0.347
dist_to_goal	1.911 (3.448)	4.230 (3.996)	**1.030 (2.028)**	1.462 (2.908)	4.106 (4.045)
SPL	0.627 (0.412)	0.293 (0.436)	**0.754 (0.329)**	0.673 (0.371)	0.314 (0.438)
SoftSPL	0.644 (0.338)	0.359 (0.381)	**0.748 (0.266)**	0.675 (0.308)	0.370 (0.383)
num_steps	208.792	381.020	**145.561**	182.773	361.389

Table 5.6: Results of comparison between OccAnt and ANS on point-goal driven exploration.

Table 5.6 displays the results of the comparison done on the 14 environments described in Section 5.1.1. In parentheses, we report also the standard deviation (the standard deviation for success_rate and num_steps is not reported because is almost uninformative) The table provides us with the information that OccAnt (depth) obtains sensibly better

results than the best ANS configuration i.e., ANS (depth). OccAnt (depth) obtains the best values in all the metrics considered. It is interesting to note that while in exploration for map building task OccAnt (rgb) and ANS (depth) obtain very similar results in all the metrics (Section 5.3.1), in point-goal driven exploration task, OccAnt (depth) obtains noticeable better results. This improvement is achieved thanks to the presence of the occupancy anticipation module, which allows the OccAnt agent to reach the goal faster. The results obtained by OccAnt (rgbd) show that also in this case, like in Section 5.3, OccAnt (rgbd) pretrained models loaded by the authors on the Github page have some kind of problem when applied to new environments. In point-goal driven exploration task, contrary to what happens in exploration for map building task, with higher average values comes lower standard deviation values.

5.4.2. DRL tests

In order to understand how DRL algorithm works and how it can be compared with ANS or OccAnt algorithms we have run different tests. First of all we have run a training for the DRL algorithm on the original Gazebo random environments (shown in Section 5.1.2). This training lasted about 12 hours and is enough to obtain 100% success rate during evaluation in Gazebo environments. In order to understand the behavior in more complex environments like the ones proposed by Gibson dataset, we have tested DRL algorithm (trained on Gazebo) in Habitat simulator through ROS_X_Habitat in Swormville environment using the episodes proposed by Gibson dataset (the same used in the previous tests). Results in this case are sensibly worse. In Figure 5.20 we can see the paths followed by the agent in different evaluation episodes. In all of episodes presented in the figure, the agent has some difficulties in moving to the goal and trying to get out of the rooms.

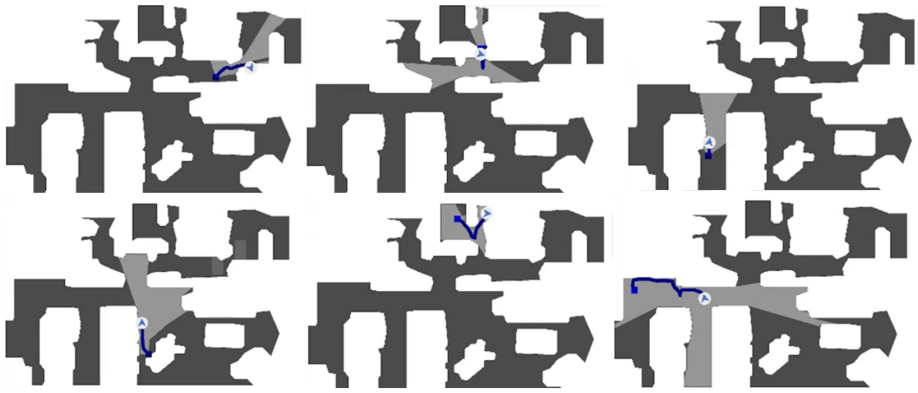


Figure 5.20: Paths followed by DRL agent in Habitat simulator in first test.

One possible explanation for this behavior is the fact that the robot is originally trained in an environment with only one square-shaped room, no corridor, nor complex elements. Analyzing the start and goal position in case of success it appears that the agent is able to reach the goal only when start and goal positions are very close (probably in the same room), being the set-up similar to the one in the DRL environments. In another test, we have trained DRL algorithm for 317 episodes in the Habitat simulator. The evaluation is done in Swormville environment.

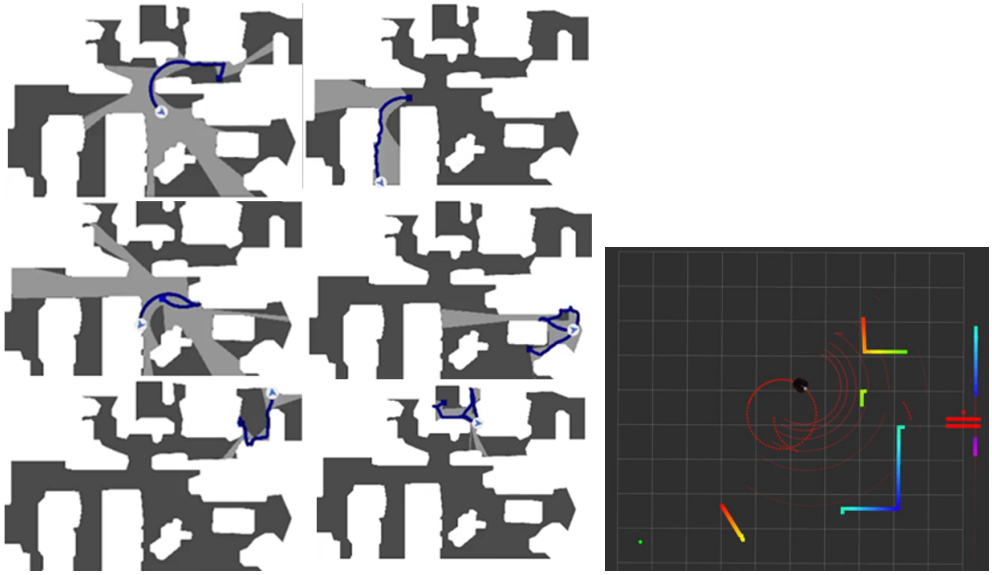


Figure 5.21: Paths followed by DRL agent in Habitat simulator in second test (left), path followed in DRL environment (right).

In Figure 5.21, on the left, we can see examples of the path followed in the Habitat simulator environment during the evaluation. The robot is now able to cover more space than the robot simply trained in the DRL environments. A strange behavior appears when we test these weights learned in the Habitat simulator back in Gazebo DRL environment: as can be seen from Figure 5.21, on the right, the robot moves in circle and doesn't try to reach the goal. One possible explanation is the fact that on Habitat there are a lot of obstacles and consequently a lot of collisions. Because of the negative reward obtained with collision the robot has learned a policy that prefers not colliding over reaching the goal. We have come to this conclusion also by looking at the training videos. In the first episodes, the robot collides a lot trying to reach the goal, but then in the subsequent episodes, it collides much less. At this point, we have compared the results of the two different weights obtained from the training in Gazebo and Habitat simulator.

	Trained on Habitat	Trained on Gazebo
succes_rate	0.2532211268	**0.26766056338**
dist_to_goal	2.2001	**1.964968772**
num_steps	**304.9859**	555.9014085
avg_reward	**-14.269613**	-127.939993
avg_col	0.676056	**0.507042**

Table 5.7: Comparison of the results on point-goal driven exploration in Habitat simulator between DRL algorithm trained in Habitat and in Gazebo simualtor.

Table 5.7 confirms what we have already described. The algorithm trained on Gazebo has the best succes_rate and distance to goal values, but the worst avg_reward and num_steps. These two last metrics, in particular, are very bad when compared to the ones obtained by the algorithm trained on Habitat. The sensibly better value of avg_reward obtained by the algorithm trained on Habitat shows us that the algorithm has learned a policy that can obtain better reward values when deployed in the Habitat environment with respect to the policy trained in Gazebo. However, in both cases, there are a lot of collisions and avg_reward is always negative, meaning that the reward function as thought for DRL environments may not be so effective in Habitat environments. With the analysis done until this moment, we have highlighted the importance and difficulty of building a reward function able to provide consistent results even in environments really different from the ones where the algorithm has been trained. This element of difficulty is not present using classical algorithms and should be carefully considered. Another aspect that should be taken into consideration is how to write a distance function. The original DRL code uses Euclidean distance for computing the distance between start and goal positions and deciding accordingly if the goal is reached or not (dist_to_goal < 0.3 m). ANS and OccAnt otherwise use obstacle-free path distance.

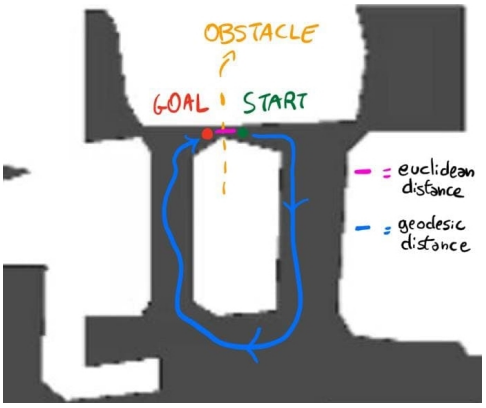


Figure 5.22: Euclidean vs. obstacle-free path distance in complex Habitat environments.

In simple environments like those of DRL, using the Euclidean distance may be an appropriate choice, but in complex environments like Habitat photorealistic ones may be not the right choice. A possible situation is the one shown in Figure 5.22, where start-goal Euclidean distance is lower than 0.3 m, but the real obstacle-free path is a way longer. As described in Section 4.2.1, we have implemented the possibility to use the obstacle-free path distance also in the DRL algorithm to compare it with ANS and OccAnt.
SPL and SoftSPL metrics can be computed only with the obstacle-free path distance value and because of that are shown only in the tests that use obstacle-free path distance values.
As done before we have compared the algorithm trained in Gazebo and Habitat, using the obstacle-free path distance, comparing over 213 episodes in 3 different environments.

	Trained in Gazebo obstacle-free path distance	**Trained in Habitat obstacle-free path distance**	**Trained in Gazebo Euclidean distance**	**Trained in Habitat Euclidean distance**
succes_rate	**0.019**	0.0	0.2532211268	**0.26766056338**
dist_to_goal	8.142	**6.827**	2.2001	**1.964968772**
num_steps	**498.648**	1275.817	**304.9859**	555.9014085
avg_reward	**-177.3473593**	-210.9696753	**-14.269613**	-127.939993
avg_col	1.1971833333	**0.5023486667**	0.676056	**0.507042**
SPL	**0.016**	0.0	X	X
SoftSPL	**0.081**	0.0205	X	X

Table 5.8: DRL obstacle-free path and Euclidean distance comparison with training in Gazebo environments and Habitat environments.

As can be clearly seen from the Table 5.8, results are worse than those obtained with Euclidean distance.
The algorithm trained in the Habitat simulator obtains better results in dist_to_goal and avg_col, but the agent behavior still prefers to avoid obstacles than reach the goal. This strengthens our idea that the reward function must be changed in order to fit the requirements of complex environments. In this case, the agent trained in the Habitat simulator doesn't obtain better results in avg_reward, but this can be referred to the fact that the algorithms in the Habitat simulator are usually trained on a big number of episodes with DD-PPO, while here we are training it with a single (not so powerful) GPU for only relatively few episodes (780).
The results obtained in this section are very low when they are compared to ANS and OccAnt results obtained in Section 5.4.1. The tests done on DRL algorithm in Habitat environments suggest that probably the reward function should be adapted to better fit the requirements in complex environments, preferring the exploration of the environments over the avoidance of having collisions. The quick training done with DRL in the Gazebo

simulator is more than enough to obtain a 100% success rate on that simple domain, but it proved to not generalize well using complex environments with multiple rooms and long and narrow corridors. DRL agent in the original setting is not rewarded for trying to reach the goal exploring different rooms to find the right path but only for reaching the goal in a simple room with a fixed dimension and few obstacles with a regular shape. While using the Euclidean distance in simple environments is not a big problem, using it in a setting with multiple objects may lead to situations where the goal is not really reached. As just suggested, one of the most likely causes of the differences between performance in algorithms is to be found in the diversities between training environments and settings. While DRL algorithm from [60] is trained with an Nvidia GTX 1080 GPU for only 8 hours to obtain good results in the Gazebo environments, ANS and OccAnt are trained with DD-PPO on configurations with multiple powerful GPUs (8 GPUs and 16/32GB memory per GPU) acquiring a way higher level of experience in exploration and goal driven navigation. As also shown in Section 4.1.4 and Figure 4.1, Habitat simulator focuses on speed, gaining high "frame per seconds" performance. This means that at the same time, algorithms trained in the Habitat simulator can acquire more experience. The quality of acquired experience proved to be very important in order to let the agent generalize well when deployed in unseen environments.
Another cause of the differences is the fact that DRL code uses a velodyne laser, while in the comparison we have replaced its readings with the laser readings provided ROS-X-Habitat, but this element shouldn't have a higher impact.
The kinds of issues related to the different computation power available for training are not present when we consider classical algorithms. Because of this reason, in the next section we compare ANS, OccAnt, and the classical algorithm in order to understand if the higher computation demand allows to obtain better performance with Deep Learning algorithms.

5.4.3. Point-goal driven exploration ANS vs. OccAnt vs. classical algorithm

In this section, we have run the comparison between ANS, OccAnt, and the classical algorithm (implemented with ROS Gmapping and move_base packages) on 3 environments of different dimensions: Elmira, Swormville, and Cantwell (sorted in ascending order by area).
In every environment, we have run 30 episodes with different starting and goal locations. ANS and OccAnt are tested in the Habitat simulator, while the classical algorithm because of the invisible textures and holes is tested in 2D maps in Stage simulator, as done

during exploration for map building comparison. As also shown in Section 5.3.3, exploration results in 3D and 2D are very similar and so the comparison will be fair. Because of the fact that SPL and SoftSPL are metrics considered only in the Habitat simulator, we have not considered them here. In this comparison, we have used success_rate and path_length as metrics. In Section 5.4.1 the results of the comparison show a high level of success and because of this, it is interesting to understand if a classical algorithm can obtain similar values with a comparable path_length.
In Section 5.3.1 we have shown that considering only area_seen as metric without the time required to visit that area doesn't provide a fair comparison. In Section 5.3.3 we have also seen that the exploration capabilities between classical and Deep Learning algorithms can be compared taking into consideration the path_lenght required to explore the area. In this section, we consider not only success_rate as a relevant metric for point-goal driven exploration, but also path_length. We use this metric because our focus is on understanding not only if all the three algorithms are able to reach the point-goals but also which of the three algorithms manages to travel fewer meters.
In Tables 5.9 and 5.10 are reported the values of the metrics obtained from the episodes run in the different environments. Because of the results obtained in the previous section we have considered only ANS (depth) and OccAnt (depth), the implementation of the two algorithms that obtain the best values. We have not considered DRL algorithm because, as seen in Section 5.4.2, its results are really low when compared to OccAnt and ANS.
The path_lenght values of all the episodes are shown in Appendix A.4.

	classical	**OccAnt (depth)**	**ANS (depth)**
	best path_length	**best path_length**	**best path_length**
Elmira	**21**	3	6
Swormville	**19**	7	4
Cantwell	**19**	10	1
Total	**59**	20	11

Table 5.9: Number of best path_length episodes in classical algorithm, OccAnt (depth), and ANS (depth).

	classical	OccAnt (depth)	ANS (depth)
	success_rate	**success_rate**	**success_rate**
Elmira	**0.967**	0.900	0.800
Swormville	**0.967**	0.900	0.767
Cantwell	**0.967**	0.833	0.600
Total	**0.967**	0.878	0.722

Table 5.10: success_rate comparison between classical algorithm, OccAnt (depth), and ANS (depth).

Results of Table 5.9 show that the classical algorithm, no matter what is the dimension of the environment, performs better: it has the best path_length in most of the episodes. Only in a few cases, results are significantly worse (as can be seen in Appendix A.4). Classical algorithm bad results are probably due to the fact that in 2D implementation all the objects in the environments are flattened on the ground and different paths can be identified by the algorithm.
In every environment, the classical algorithm also has more success episodes (as shown in Table 5.10). The classical algorithm fails only 3 times: in one case the agent collides at the beginning of the episode while in the other two cases the agent isn't able to find the right path.
Even if OccAnt (depth) with occupancy anticipation module proved to perform better with respect to ANS (depth) (Section 5.4.1), there is no clear advantage over the classical algorithm neither in success_rate nor in path_length metric. Success_rate values shown in Table 5.10, concerning OccAnt (depth) and ANS (depth), are comparable to the ones obtained in the previous section (Table 5.6) in 994 episodes and therefore 90 episodes are enough to obtain a fair comparison between classical algorithm, ANS, and OccAnt. While the success_rate in the classical algorithm is constant and is not conditioned by the size of the environment, both OccAnt and ANS have lower values as the size of the environment increases. The classical algorithm not only doesn't need expensive training in these kinds of environments, but it also outperforms the Deep Learning algorithms analyzed in all the metrics considered. In addition, the classical algorithm already proved to perform well in the real world, while Deep Learning algorithms need to be carefully tested.

5.4.4. Results anaysis

Results of the comparison done between OccAnt and ANS in point-goal driven exploration task (Section 5.4.1) show that the presence of the occupancy anticipation module allows OccAnt to achieve higher performance when compared to ANS. However, the occupancy anticipation module, a Deep Learning network that requires extensive training, doesn't give any clear advantage in exploration for map building task (Sections 5.3.1 and 5.3.2), and because of the computational and energy resources it needs, it should be considered whether this module is worth using.
DRL bad results from Section 5.4.2 highlight the difficulties of writing a strong reward function, which is also able to generalize well in unknown environments. The agent trained in Gazebo simple environments has some problems in exploring Habitat complex environments with multiple rooms and corridors. DRL agent has proved to not perform well also when the training is done in Habitat environments, probably because of the relatively few episodes (780) of training. The tests done also suggest that probably the reward function should be adapted to better fit the requirements in complex environments.
In Section 5.4.3, we have shown that even if OccAnt (depth) with occupancy anticipation module proved to perform better with respect to ANS (depth), there is no clear advantage over the classical algorithm neither in success_rate nor in path_length metric.
All the difficulties encountered by the Deep Learning algorithms described in this chapter should make us consider whether to use these techniques for the point-goal driven exploration task, considering that the classical algorithms have already proven to provide the optimal solution.

6 | Conclusion and future work

In this thesis, we have compared classical and Deep Learning algorithms on two different tasks: exploration for map building and point-goal driven exploration in order to understand what are the downsides and upsides of the algorithms proposed. Every task has been evaluated using task-specific metrics in common environments.
The comparison is performed using the Habitat simulator, ROS-X-Habitat framework, Gazebo simulator, and Stage simulator.
OccAnt [97] and ANS [48] have been compared on a big number of episodes and environments on the Gibson dataset (Section 5.3.1 and Section 5.3.2), showing what is the improvement given by the occupancy anticipation module introduced in OccAnt architecture. While their results are comparable in exploration for map building task, OccAnt achieves better results in point-goal driven exploration task. The occupancy anticipation module doesn't help to build more accurate maps, but it allows better performance when the agent has to reach a specific point-goal in space.
OccAnt and ANS have also been tested in exploration tasks on both noisy and noise free simulations. This evaluation has shown that ANS compared to OccAnt explores a little less, but with higher accuracy.
The results of the comparison done using Habitat simulator and environments on the Gibson dataset between frontier exploration [8], OccAnt, and ANS on exploration for map building task (Section 5.3.3) show that they all have comparable performances on the average. OccAnt and ANS, however, require a very long training time with big computational power and energy resources involved [6]. On the other side, the classical algorithm with frontier exploration can be simply implemented on an agent and doesn't have to deal with all the issues regarding training or ability to generalize to an unknown environment, because its performance doesn't depend on the number and the quality of training environments. In Section 5.3.4 we have also compared the decision-making time required by these three algorithms and the results show that both OccAnt and ANS outperform frontier exploration. Frontier exploration decision-making time, however, is still low (0.095 s) and fully compatible with real world situations. In Section 5.4.3 we have compared ANS and OccAnt with a classical algorithm showing that also on point-goal driven exploration

task the results are very similar.
For the learning algorithms compared and for most of the literature considered, there is little information about what is the real cost of training and hyperparameter-tuning. Training is a very long activity (days and in some cases weeks) done on high-end computers and its cost should be made more clear.
The development of a working reward function and the search required to obtain good hyperparameters are examples of other costs that should be taken into consideration in the future in order to correctly compare Deep Learning and classical exploration algorithms. The tradeoff between engineering costs and performance obtained should be really considered in order to understand if the effort required to develop Deep Learning algorithms is worth. An example of how the duration of the training can have an impact on the quality of exploration policy learned is provided in Section 5.4.2 where it has been shown that DRL algorithm [60] trained a little time in complex environments doesn't behave well compared to algorithms trained for a longer time with powerful GPUs.
In most of the literature considered and in the algorithms compared, the authors proposing Deep Learning algorithms omit to consider classical approaches like frontier exploration as a baseline. In many cases, in fact, authors only consider other learning algorithms and, in some cases, the metrics considered are relevant only for Deep Learning. Knowing that a learning algorithm performs better compared to another learning algorithm doesn't give us any information about its performance when compared to classical state-of-the-art algorithms.
Metrics that don't lead to confusion should be taken into consideration in the future, in order to be able to correctly compare learning and classical algorithms. In OccAnt and ANS, for example, the map_accuracy metrics doesn't take into consideration the path_length required to obtain that mapping accuracy. As seen, when we take into consideration this metric, we can compare Deep Learning and classical algorithms. In DRL the avg_reward metric provided by the authors of the original implementation is not even useful in order to compare it to the other two learning algorithms (OccAnt and ANS).
Our comparison highlights the fact that performance of OccAnt, ANS, and the classical algorithm in complex simulation environments is comparable and because of the fact that Deep Learning algorithms are known to exploit imperfections in order to obtain better performance, it would be useful to know if these results can also be achieved in the real world. The real world environments where the algorithms are tested should be complex (with a lot of rooms and obstacles) and not simple like the one reported in [73].
The difficulty of writing a strong reward function, which is also able to generalize well in unknown environments has been discussed theoretically in Section 2.4.1 and experimentally tested in Section 5.4.2. Here we can see how the DRL algorithm performs well in the

simple environments where it is tested, but when it is compared with ANS and OccAnt in more complex environments its results are significantly lower. Furthermore, most of Deep Learning algorithms don't provide interpretability, contrary to classical algorithms. Because of this reason, we can't be really sure of what is the real problem when we deploy DRL in a complex environment.

DRL's lower performance also stresses out the importance of acquiring experience for a Deep Learning algorithm. A good solution in order to avoid the problems identified with DRL may be to train the algorithm in different simulators in an effort to acquire as much experience as possible before deploying the agent in the real world. Training of the Deep Learning algorithms on different simulators may also avoid them to learn the imperfections of a particular simulator. In all the papers examined, the training and the testing in the simulator or in the real world are considered separate phases. This disjunction doesn't allow an agent to also learn from experience in the real world once it has been trained (necessarily) in a simulator. As a future direction, it may be useful to consider approaches where the agent is also able to learn from experience in the real world which, as already mentioned, is naturally very different from all the simulators considered.

Bibliography

[1] Ai2thor simulator. https://ai2thor.allenai.org/. Last accessed: 01/02/2022.

[2] AI Habitat. https://aihabitat.org/. Last accessed: 06/01/2022.

[3] Habitat-lab. https://github.com/facebookresearch/habitat-lab. Last accessed: 06/01/2022.

[4] Habitat-sim. https://github.com/facebookresearch/habitat-sim. Last accessed: 06/01/2022.

[5] AI Habitat youtube. https://www.youtube.com/watch?v=L9GuINYhmZI&list=PLGywud_-HlCORC0c4uj97oppQrGiB6JNy. Last accessed: 06/01/2022.

[6] Ans training time. https://github.com/devendrachaplot/Neural-SLAM/issues/30. Last accessed: 18/04/2022.

[7] Drl robot navigation. https://github.com/reiniscimurs/DRL-robot-navigation. Last accessed: 06/01/2022.

[8] Explore lite - frontier exploration. http://wiki.ros.org/explore_lite. Last accessed: 01/02/2022.

[9] Gazebo simulator. http://gazebosim.org/. Last accessed: 16/01/2022.

[10] Gibson dataset. http://gibsonenv.stanford.edu/database/. Last accessed: 06/01/2022.

[11] Utilities for Gibson environment. https://github.com/micheleantonazzi/gibson-env-utilities. Last accessed: 12/03/2022.

[12] Gmapping - SLAM. http://wiki.ros.org/gmapping. Last accessed: 01/02/2022.

[13] Hm3d. https://aihabitat.org/datasets/hm3d/. Last accessed: 06/01/2022.

[14] Lifull home's dataset. https:/F/www.nii.ac.jp/dsc/idr/en/lifull/. Last accessed: 01/02/2022.

[15] Habitat noise. https://github.com/facebookresearch/habitat-sim/pull/89. Last accessed: 16/02/2022.

[16] Habitat sliding problem. https://github.com/facebookresearch/habitat-sim/pull/439. Last accessed: 01/02/2022.

[17] Explainable AI. https://becominghuman.ai/its-magic-i-owe-you-no-explanation-explainableai-43e798273a08. Last accessed: 25/05/2022.

[18] Keras resnet. https://github.com/raghakot/keras-resnet/blob/master/resnet.py. Last accessed: 24/01/2022.

[19] Pyrobot. http://www.locobot.org/. Last accessed: 21/01/2022.

[20] Matterport3d. https://niessner.github.io/Matterport/. Last accessed: 06/01/2022.

[21] movebase. http://wiki.ros.org/move_base. Last accessed: 14/03/2022.

[22] Occupancy anticipation. https://github.com/facebookresearch/OccupancyAnticipation. Last accessed: 16/01/2022.

[23] ROS. https://ros.org. Last accessed: 07/01/2022.

[24] ROS nodes. http://wiki.ros.org/Nodes. Last accessed: 08/06/2022.

[25] ROS packages. http://docs.ros.org/en/independent/api/rospkg/html/packages.html. Last accessed: 21/06/2022.

[26] ROS stage simulator. http://wiki.ros.org/stage_ros. Last accessed: 01/02/2022.

[27] Rviz. https://github.com/ros-visualization/rviz. Last accessed: 07/01/2022.

[28] Replica. https://github.com/facebookresearch/Replica-Dataset. Last accessed: 06/01/2022.

[29] ROS-X-Habitat. https://github.com/ericchen321/ros_x_habitat. Last accessed: 16/01/2022.

[30] SLAM. http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf. Last accessed: 17/02/2022.

[31] Trajectory rollout. `http://wiki.ros.org/base_local_planner#TrajectoryPlannerROS`. Last accessed: 07/06/2022.

[32] VAE and CVAE. `https://www.cs.hhu.de/fileadmin/redaktion/Fakultaeten/Mathematisch-Naturwissenschaftliche_Fakultaet/Informatik/Dialog_Systems_and_Machine_Learning/052020_vae.pdf`. Last accessed: 31/01/2022.

[33] Vizdoom simulator. `https://github.com/mwydmuch/ViZDoom`. Last accessed: 01/02/2022.

[34] Fmm path planning. `https://jvgomez.github.io/files/pubs/fm2star.pdf`. Last accessed: 22/05/2022.

[35] P. Abbeel and A. Y. Ng. Apprenticeship learning via Inverse Reinforcement Learning. In *Proceedings of the twenty-first International Conference on Machine learning*, pages 1–8, 2004.

[36] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *Proceedings of the 2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[37] F. Amigoni. Experimental evaluation of some exploration strategies for mobile robots. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation*, pages 2818–2823. IEEE, 2008.

[38] F. Amigoni, M. Luperto, and V. Schiaffonati. Toward generalization of experimental results for autonomous robots. *Robotics and Autonomous Systems*, 90:4–14, 2017.

[39] O. Asraf and V. Indelman. Experience-based prediction of unknown environments for enhanced belief space planning. In *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6781–6788, 2020.

[40] A. Aydemir, P. Jensfelt, and J. Folkesson. What can we learn from 38,000 rooms? reasoning about unexplored space in indoor environments. In *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4675–4682, 2012.

[41] S. Bai, F. Chen, and B. Englot. Toward autonomous mapping and exploration for mobile robots through deep supervised learning. In *Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2379–2384, 2017.

[42] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[43] F. Bissmarck, M. Svensson, and G. Tolt. Efficient algorithms for next best view evaluation. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5876–5883, 2015.

[44] A. Buitrago-Martínez, R. F. De la Rosa, and F. Lozano-Martínez. Hierarchical reinforcement learning approach for motion planning in mobile robotics. In *Proceedings of the 2013 Latin American Robotics Symposium and Competition*, pages 83–88, 2013.

[45] Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[46] J. A. Caley, N. R. Lawrance, and G. A. Hollinger. Deep learning of structured environments for robot search. *Autonomous Robots*, 43(7):1695–1714, 2019.

[47] D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi. Autonomous exploration for search and rescue robots. *WIT Transactions on the Built Environment*, 94:1–10, 2007.

[48] D. S. Chaplot, D. Gandhi, S. Gupta, A. Gupta, and R. Salakhutdinov. Learning to explore using active neural slam. *arXiv preprint arXiv:2004.05155*, 2020.

[49] D. S. Chaplot, D. P. Gandhi, A. Gupta, and R. R. Salakhutdinov. Object goal navigation using goal-oriented semantic exploration. *Advances in Neural Information Processing Systems*, 33:4247–4258, 2020.

[50] D. S. Chaplot, H. Jiang, S. Gupta, and A. Gupta. Semantic curiosity for active visual learning. In *Proceedings of the European Conference on Computer Vision*, pages 309–326, 2020.

[51] D. S. Chaplot, M. Dalal, S. Gupta, J. Malik, and R. Salakhutdinov. Seal: Self-supervised embodied active learning using exploration and 3d consistency. *arXiv preprint arXiv:2112.01001*, 2021.

[52] C. Chen, S. Majumder, Z. Al-Halah, R. Gao, S. K. Ramakrishnan, and K. Grauman. Learning to set waypoints for audio-visual navigation. *arXiv preprint arXiv:2008.09622*, 2020.

[53] F. Chen, J. D. Martin, Y. Huang, J. Wang, and B. Englot. Autonomous exploration under uncertainty via deep reinforcement learning on graphs. In *Proceedings of*

the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 6140–6147, 2020.

[54] G. Chen, H. Yang, and I. M. Mitchell. Ros-x-habitat: Bridging the ros ecosystem with embodied ai. *arXiv preprint arXiv:2109.07703*, 2021.

[55] T. Chen, S. Gupta, and A. Gupta. Learning exploration policies for navigation. *arXiv preprint arXiv:1903.01959*, 2019.

[56] W. Chen, T. Qu, Y. Zhou, K. Weng, G. Wang, and G. Fu. Door recognition and deep learning algorithm for visual based robot navigation. In *Proceedings of the 2014 IEEE International Conference on Robotics and Biomimetics (robio 2014)*, pages 1793–1798, 2014.

[57] Y.-h. Chen, I. L. Moreno, T. Sainath, M. Visontai, R. Alvarez, and C. Parada. Locally-connected and convolutional neural networks for small footprint speaker recognition. In *Proceedings of the Sixteenth Annual Conference of the International Speech Communication Association*, pages 1–5, 2015.

[58] S. Chernova and M. Veloso. Confidence-based policy learning from demonstration using gaussian mixture models. In *Proceedings of the 6th International joint Conference on Autonomous Agents and Multiagent Systems*, pages 1–8, 2007.

[59] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[60] R. Cimurs, I. H. Suh, and J. H. Lee. Goal-driven autonomous exploration through deep reinforcement learning. *IEEE Robotics and Automation Letters*, 7(2):730–737, 2022. doi: 10.1109/LRA.2021.3133591.

[61] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[62] A. Das, S. Datta, G. Gkioxari, S. Lee, D. Parikh, and D. Batra. Embodied question answering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–10, 2018.

[63] S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *Proceedings of the International Conference on Machine Learning*, pages 1587–1596, 2018.

[64] H. H. González-Banos and J.-C. Latombe. Navigation strategies for exploring indoor environments. *The International Journal of Robotics Research*, 21(10-11):829–848, 2002.

[65] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the International Conference on Machine Learning*, pages 1861–1870, 2018.

[66] S. Y. Hayoun, E. Zwecher, E. Iceland, A. Revivo, S. R. Levy, and A. Barel. Integrating deep-learning-based image completion and motion planning to expedite indoor mapping. *arXiv preprint arXiv:2011.02043*, 2020.

[67] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[68] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[69] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, pages 3215–3222, 2018.

[70] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[71] S. Jung and D. H. Shim. Mapless navigation: Learning uavs motion for exploration of unknown environments. *arXiv preprint arXiv:2110.01747*, 2021.

[72] N. Justesen, P. Bontrager, J. Togelius, and S. Risi. Deep learning for video game playing. *IEEE Transactions on Games*, 12(1):1–20, 2019.

[73] A. Kadian, J. Truong, A. Gokaslan, A. Clegg, E. Wijmans, S. Lee, M. Savva, S. Chernova, and D. Batra. Sim2real predictivity: Does evaluation in simulation predict real-world performance? *IEEE Robotics and Automation Letters*, 5(4): 6670–6677, 2020.

[74] Y. Katsumata, A. Kanechika, A. Taniguchi, L. E. Hafi, Y. Hagiwara, and T. Taniguchi. Map completion from partial observation using the global structure of multiple environmental maps. *arXiv preprint arXiv:2103.09071*, 2021.

[75] K. Katyal, K. Popek, C. Paxton, P. Burlina, and G. D. Hager. Uncertainty-aware occupancy map prediction using generative networks for robot navigation. In *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*, pages 5453–5459, 2019.

[76] D. I. Koutras, A. C. Kapoutsis, A. A. Amanatiadis, and E. B. Kosmatopoulos. Marsexplorer: Exploration of unknown terrains via deep reinforcement learning and procedurally generated environments. *arXiv preprint arXiv:2107.09996*, 2021.

[77] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[78] G. Kumar, N. S. Shankar, H. Didwania, R. Roychoudhury, B. Bhowmick, and K. M. Krishna. Gcexp: Goal-conditioned exploration for object goal navigation. In *Proceedings of the 2021 30th IEEE International Conference on Robot & Human Interactive Communication (RO-MAN)*, pages 123–130, 2021.

[79] T. Li, D. Ho, C. Li, D. Zhu, C. Wang, and M. Q.-H. Meng. Houseexpo: A large-scale 2D indoor layout dataset for learning-based algorithms on mobile robots. In *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5839–5846, 2020.

[80] M. Luperto and F. Amigoni. Extracting structure of buildings using layout reconstruction. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, pages 652–667, 2018.

[81] M. Luperto, V. Arcerito, and F. Amigoni. Predicting the layout of partially observed rooms from grid maps. In *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*, pages 6898–6904, 2019.

[82] M. Luperto, L. Fochetta, and F. Amigoni. Exploration of indoor environments through predicting the layout of partially observed rooms. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 836–843, 2021.

[83] S. Macenski. On use of the slam toolbox: A fresh (er) look at mapping and localization for the dynamic world. In *Proceedings of the ROSCon*, pages 1–17, 2019.

[84] Manolis Savva*, Abhishek Kadian*, Oleksandr Maksymets*, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh, and D. Batra. Habitat: A

Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9339–9347, 2019.

[85] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[86] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine learning*, pages 1928–1937, 2016.

[87] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 1928–1937, 2016.

[88] A. Murali, T. Chen, K. V. Alwala, D. Gandhi, L. Pinto, S. Gupta, and A. Gupta. Pyrobot: An open-source robotics framework for research and benchmarking. *arXiv preprint arXiv:1906.08236*, 2019.

[89] T. Nagarajan and K. Grauman. Learning affordance landscapes for interaction exploration in 3d environments. *arXiv preprint arXiv:2008.09241*, 2020.

[90] M. Narasimhan, E. Wijmans, X. Chen, T. Darrell, D. Batra, D. Parikh, and A. Singh. Seeing the un-scene: Learning amodal semantic maps for room navigation. In *Proceedings of the European Conference on Computer Vision*, pages 513–529, 2020.

[91] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE Access*, 7:19143–19165, 2019.

[92] F. Niroui, K. Zhang, Z. Kashino, and G. Nejat. Deep reinforcement learning robot for search and rescue applications: Exploration in unknown cluttered environments. *IEEE Robotics and Automation Letters*, 4(2):610–617, 2019.

[93] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[94] M. Pak and S. Kim. A review of deep learning in image recognition. In *2017*

4th International Conference on Computer Applications and Information Processing Technology (CAIPT), pages 1–3. IEEE, 2017.

[95] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8 (7):579–588, 2009.

[96] W. Qi, R. T. Mullapudi, S. Gupta, and D. Ramanan. Learning to move with affordance maps. *arXiv preprint arXiv:2001.02364*, 2020.

[97] S. K. Ramakrishnan, Z. Al-Halah, and K. Grauman. Occupancy anticipation for efficient exploration and navigation. In *Proceedings of the European Conference on Computer Vision*, pages 400–418, 2020.

[98] E. Remolina and B. Kuipers. Towards a general theory of topological maps. *Artificial Intelligence*, 152(1):47–104, 2004.

[99] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241, 2015.

[100] M. Saroya, G. Best, and G. A. Hollinger. Online exploration of tunnel networks leveraging topological cnn-based world predictions. In *Proceedings of the 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6038–6045, 2020.

[101] N. Savinov, A. Dosovitskiy, and V. Koltun. Semi-parametric topological memory for navigation. *arXiv preprint arXiv:1803.00653*, 2018.

[102] M. Savva, A. X. Chang, A. Dosovitskiy, T. Funkhouser, and V. Koltun. Minos: Multimodal indoor simulator for navigation in complex environments. *arXiv preprint arXiv:1712.03931*, 2017.

[103] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[104] J. A. Sethian. Fast marching methods. *SIAM review*, 41(2):199–235, 1999.

[105] S. K. Sharma and S. Kumar. Comparative analysis of Manhattan and Euclidean distance metrics using A* algorithm. *Journal of Research in Engineering and Applied Sciences*, 1(4):196–198, 2016.

[106] R. Shrestha, F.-P. Tian, W. Feng, P. Tan, and R. Vaughan. Learned map prediction

for enhanced mobile robot exploration. In *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*, pages 1197–1204, 2019.

[107] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[108] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587): 484–489, 2016.

[109] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[110] A. J. Smith and G. A. Hollinger. Distributed inference-based multi-robot exploration. *Autonomous Robots*, 42(8):1651–1668, 2018.

[111] Y. Song, Y. Hu, J. Zeng, C. Hu, L. Qin, and Q. Yin. Towards efficient exploration in unknown spaces: A novel hierarchical approach based on intrinsic rewards. In *Proceedings of the 2021 6th International Conference on Automation, Control and Robotics Engineering (CACRE)*, pages 414–422, 2021.

[112] C. Stachniss, J. J. Leonard, and S. Thrun. Simultaneous localization and mapping. In *Springer Handbook of Robotics*, pages 1153–1176. Springer, 2016.

[113] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[114] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[115] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, M. Mukadam, D. Chaplot, O. Maksymets, A. Gokaslan, V. Vondrus, S. Dharur, F. Meier, W. Galuba, A. Chang, Z. Kira, V. Koltun, J. Malik, M. Savva, and D. Batra. Habitat 2.0: Training home assistants to rearrange their habitat. *arXiv preprint arXiv:2106.14405*, 2021.

[116] F. Torabi, G. Warnell, and P. Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[117] R. Vaughan. Massively multi-robot simulation in stage. *Swarm intelligence*, 2(2): 189–208, 2008.

[118] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, and D. Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv preprint arXiv:1911.00357*, 2019.

[119] Y. Wu, Y. Wu, G. Gkioxari, and Y. Tian. Building generalizable agents with a realistic and rich 3d environment. *arXiv preprint arXiv:1801.02209*, 2018.

[120] X. Xiao, D. Xu, and W. Wan. Overview: Video recognition from handcrafted method to deep learning method. In *2016 International Conference on Audio, Language and Image Processing (ICALIP)*, pages 646–651. IEEE, 2016.

[121] B. Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97.*, pages 146–151, 1997.

[122] J. Ye, D. Batra, A. Das, and E. Wijmans. Auxiliary tasks and exploration enable objectgoal navigation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 16117–16126, 2021.

[123] Z. Zheng, C. Cao, and J. Pan. A hierarchical approach for mobile robot exploration in pedestrian crowd. *IEEE Robotics and Automation Letters*, 7(1):175–182, 2021.

[124] E. Zwecher, E. Iceland, S. R. Levy, S. Y. Hayoun, O. Gal, and A. Barel. Integrating deep reinforcement and supervised learning to expedite indoor mapping. *arXiv preprint arXiv:2109.08490*, 2021.

A | Appendix A

A.1. OccAnt vs. ANS exploration for map building results - Noise free - Different environments size

In this section, we show the results of the comparison between OccAnt and ANS on exploration for map building task in small, medium, and big environments (noise free). In parentheses, we report also the standard deviation.

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	**38.761 (8.638)**	36.685 (7.761)	38.068 (8.481)
area_seen	**45.552 (9.960)**	45.160 (9.431)	45.235 (10.030)
free_space_seen	22.603 (4.985)	22.503 (4.720)	**22.873 (5.121)**
occupied_space_seen	**22.949 (6.292)**	22.656 (6.055)	22.361 (6.213)
time_per_episode	**0.865 (0.002)**	0.896 (0.002)	0.995 (0.004)
area_seen_over_time	**52.661 (10.514)**	50.402 (10.525)	45.462 (10.008)
AC/AS	0.851 (0.025)	0.812 (0.033)	**0.942 (0.030)**

Table A.1: Results of exploration for map building done in small area group environments with OccAnt and ANS in noise free simulation.

In Table A.1 are presented only the results of the comparison done on the 568 episodes of the 8 small area group environments (Section 5.1.1).

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	57.686 (17.943)	51.900 (16.387)	**61.562 (17.314)**
area_seen	66.340 (20.068)	62.584 (20.627)	**69.377 (20.772)**
free_space_seen	35.278 (11.007)	33.643 (12.050)	**37.394 (11.397)**
occupied_space_seen	31.062 (10.414)	28.940 (9.961)	**31.983 (10.762)**
time_per_episode	**0.866 (0.002)**	0.910 (0.002)	0.991 (0.005)
area_seen_over_time	**76.605 (23.173)**	68.774 (22.667)	70.007 (20.961)
AC/AS	0.870 (0.061)	0.829 (0.063)	**0.887 (0.0494)**

Table A.2: Results of exploration for map building done in medium area group environments with OccAnt and ANS in noise free simulation.

In Table A.2 are presented only the results of the comparison done on the 213 episodes of the 3 medium area group environments (Section 5.1.1).

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	60.389 (20.648)	53.172 (19.461)	**64.306 (22.540)**
area_seen	71.697 (22.969)	68.250 (22.679)	**75.593 (24.817)**
free_space_seen	43.517 (11.334)	41.958 (11.699)	**46.237 (11.915)**
occupied_space_seen	28.181 (12.106)	26.293 (11.501)	**29.355 (13.281)**
time_per_episode	**0.878 (0.002)**	0.917 (0.002)	0.968 (0.005)
area_seen_over_time	**81.659 (25.161)**	74.427 (24.732)	78.092 (25.637)
AC/AS	0.842 (0.060)	0.779 (0.053)	**0.850 (0.046)**

Table A.3: Results of exploration for map building done in big area group environments with OccAnt and ANS in noise free simulation.

In Table A.3 are presented only the results of the comparison done on the 213 episodes of the 3 big area group environments (Section 5.1.1).

A.2. OccAnt vs. ANS exploration for map building results - Noisy - Different environments size

In this section, we show the results of the comparison between OccAnt and ANS on exploration for map building task in small, medium, and big environments (noisy). In parentheses, we report also the standard deviation.

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	**35.142 (7.354)**	33.443(7.015)	34.092(7.250)
area_seen	**44.480 (8.863)**	44.084 (8.871)	**44.807 (9.213)**
free_space_seen	**22.260 (4.576)**	22.071 (4.489)	**22.704 (4.794)**
occupied_space_seen	**22.220 (5.776)**	22.013 (5.718)	**22.102 (5.852)**
time_per_episode	**0.842 (0.002)**	0.877 (0.002)	**0.843 (0.005)**
area_seen_over_time	**52.827 (10.540)**	50.267 (9.292)	**53.151 (9.579)**
AC/AS	**0.790 (0.460)**	0.758 (0.051)	0.760 (0.0569)

Table A.4: Results of exploration for map building done in small area group environments with OccAnt and ANS in noisy simulation.

In Table A.4 are presented only the results of the comparison on the 568 episodes of the 8 small area group environments (Section 5.1.1).

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	45.575 (15.184)	43.208 (12.651)	**49.424 (12.817)**
area_seen	59.261 (17.921)	56.615 (17.295)	**63.979 (17.406)**
free_space_seen	32.080 (10.334)	31.101 (10.093)	**34.993 (10.12)**
occupied_space_seen	27.026 (9.089)	25.513 (8.802)	**28.986 (8.948)**
time_per_episode	**0.843 (0.002)**	0.878 (0.002)	0.966 (0.006)
area_seen_over_time	**70.298 (21.270)**	64.482 (19.744)	66.231 (18.067)
AC/AS	**0.803 (0.077)**	0.763 (0.0687)	0.772 (0.083)

Table A.5: Results of exploration for map building done in medium area group environments with OccAnt and ANS in noisy simulation.

In Table A.5 are presented only the results of the comparison on the 213 episodes of the 3 medium area group environments (Section 5.1.1).

	ANS (depth)	ANS (rgb)	OccAnt (rgb)
map_accuracy	49.531 (16.093)	44.747 (13.646)	**52.197 (16.405)**
area_seen	63.934 (19.146)	61.846 (17.094)	**68.822 (21.414)**
free_space_seen	39.504 (9.766)	38.329 (8.828)	**42.590 (10.474)**
occupied_space_seen	24.430 (10.038)	23.517 (8.944)	**26.232 (11.451)**
time_per_episode	**0.843 (0.002)**	0.878 (0.002)	0.964 (0.003)
area_seen_over_time	**75.841 (22.651)**	70.440 (19.434)	71.392 (22.113)
AC/AS	**0.774 (0.058)**	0.726 (0.062)	0.758 (0.079)

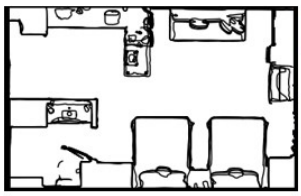
Table A.6: Results of exploration for map building done in big area group environments with OccAnt and ANS in noisy simulation.

In Table A.6 are presented only the results of the comparison on the 213 episodes of the 3 big area group environments (Section 5.1.1).

A.3. Frontier exploration vs. OccAnt vs. ANS - Small environments

In this section, we show the results of the comparison between OccAnt, ANS, and frontier exploration on exploration for map building task in two additional small environments.

A.3.1. Elmira



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	7.270	17.245
ANS (depth)	7.252	16.355
OccAnt (rgb)	7.406	17.575
Front. Exp.	7.315	//
Front. Exp. 2D	7.323	19.868

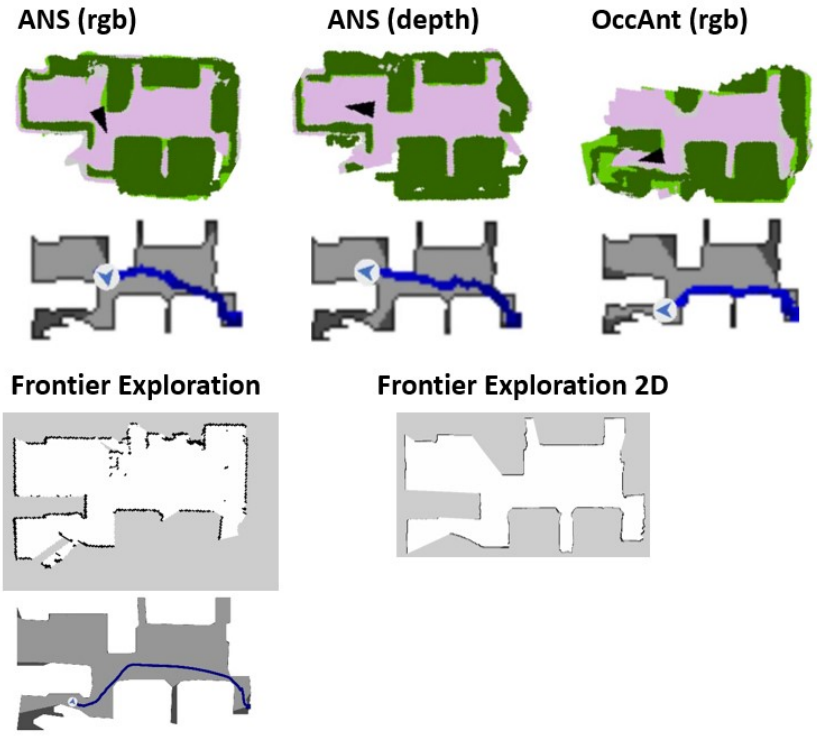
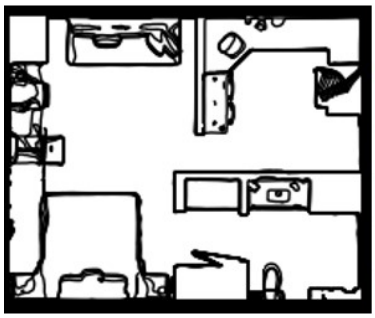


Figure A.1: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Elmira.

From maps and table in Figure A.1 we can see that with the same path_length frontier exploration in both 3D and 2D environments map more area than ANS and OccAnt algorithm.

A.3.2. Eudora



	path_length (m)	free_space_seen (m^2)
ANS (rgb)	6.684	16.577
ANS (depth)	6.748	16.300
OccAnt (rgb)	6.880	13.732
Front. Exp.	6.854	//
Front. Exp. 2D	6.756	12.864

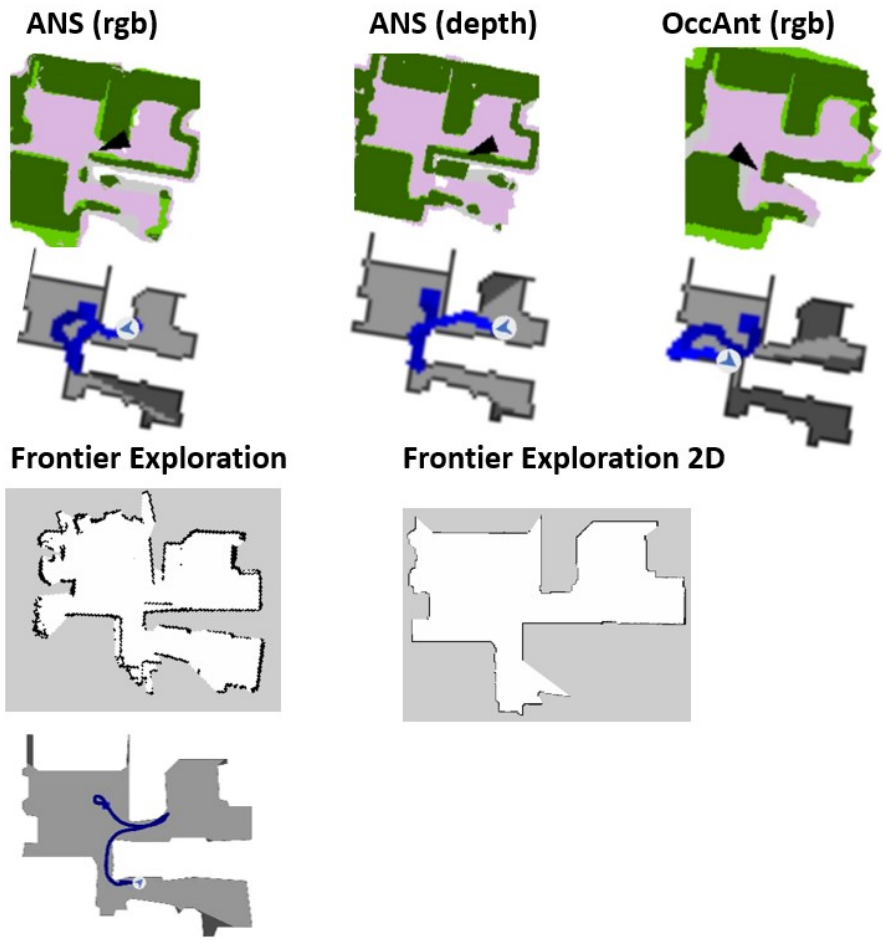


Figure A.2: Map produced and path_length of ANS (rgb), ANS (depth), OccAnt (rgb), frontier exploration, and frontier exploration 2D in Eudora.

In this case, frontier exploration 2D follows a slightly different path with respect to the one proposed by frontier exploration in 3D and so at the same path_length it still hasn't reached the last room. The room is being explored shortly after: at 7.586 m the free_space_seen is 16.214 m^2.

A.4. Point-goal driven exploration ANS vs. OccAnt vs. classical algorithm - Complete tables

In this section, we show all the path_length values of the point-goal driven exploration comparison between ANS, OccAnt, and classical algorithm.

#		classical		OccAnt (depth)		ANS (depth)
	success	path_length	success	path_length	success	path_length
0	1	**7.277**	1	7.284	1	7.396
1	1	**3.750**	0	0.193	0	0.189
2	0	0.2	1	**5.477**	1	5.492
3	1	**5.799**	1	6.330	0	3.137
4	1	**7.306**	1	7.394	1	7.638
5	1	2.580	1	2.508	1	**2.503**
6	1	**5.472**	0	0.207	0	1.006
7	1	4.417	1	**3.962**	1	4.013
8	1	**5.100**	1	5.181	1	5.151
9	1	4.330	1	4.589	1	**4.325**
10	1	**5.757**	0	4.917	0	1.513
11	1	**3.356**	1	3.557	1	3.533
12	1	**3.522**	1	3.531	1	3.598
13	1	3.950	1	5.513	1	**3.944**
14	1	**3.993**	1	5.195	1	5.237
15	1	**2.610**	1	2.665	1	4.296
16	1	**4.010**	1	4.159	1	4.241
17	1	**3.248**	1	3.251	1	3.252
18	1	**5.255**	1	5.332	1	5.619
19	1	1.638	1	1.900	1	**1.621**
20	1	**5.339**	1	5.844	1	5.592
21	1	**3.497**	1	3.574	1	3.530
22	1	**6.773**	1	7.054	1	7.731
23	1	**4.132**	1	4.248	1	4.170
24	1	6.513	1	6.951	1	**6.293**
25	1	**4.825**	1	5.775	1	7.824

26	1	3.680	1	3.806	1	**3.679**
27	1	5.214	1	**5.096**	1	5.131
28	1	**4.730**	1	5.410	0	2.958
29	1	**4.380**	1	5.348	0	10.466

Table A.7: Results of comparison (with 30 episodes) on point-goal driven exploration in Elmira between classical algorithm, OccAnt (depth), and ANS (depth).

#		classical		OccAnt (depth)		ANS (depth)
	success	path_length	success	path_length	success	path_length
0	1	**5.770**	1	6.868	1	6.227
1	1	**7.200**	1	8.011	0	2.170
2	1	**2.802**	1	3.455	1	3.052
3	1	**8.985**	1	9.886	1	10.035
4	0	19.50	1	**9.391**	0	7.527
5	1	5.920	1	**5.438**	0	3.447
6	1	**5.294**	1	5.436	1	5.659
7	1	**6.220**	0	1.465	1	6.379
8	1	5.195	1	**5.101**	1	5.283
9	1	**8.211**	1	9.196	0	3.359
10	1	**6.140**	1	6.860	1	7.330
11	1	**3.980**	1	4.007	0	3.889
12	1	**8.490**	1	8.576	1	9.235
13	1	**9.871**	1	9.937	1	12.346
14	1	4.05	1	3.838	1	**3.458**
15	1	**6.046**	1	9.920	1	6.246
16	1	2.965	1	2.992	1	**2.903**
17	1	7.887	1	**7.849**	1	10.058
18	1	**6.770**	1	7.312	1	7.948
19	1	15.145	1	**9.852**	1	11.509
20	1	16.240	1	**9.051**	0	20.951
21	1	**3.619**	1	3.972	1	3.690
22	1	**7.370**	1	7.450	1	9.962
23	1	5.900	1	5.622	1	**5.574**

24	1	8.980	1	**8.738**	1	10.904
25	1	**5.510**	0	5.671	0	1.369
26	1	**4.086**	1	4.209	1	4.582
27	1	7.120	1	6.290	1	**5.672**
28	1	**5.120**	1	5.135	1	5.309
29	1	**1.520**	0	0.514	0	0.506

Table A.8: Results of comparison (with 30 episodes) on point-goal driven exploration in Swormville between classical algorithm, OccAnt (depth), and ANS (depth).

#		**classical**		**OccAnt (depth)**		**ANS (depth)**
	success	**path_length**	**success**	**path_length**	**success**	**path_length**
0	1	**13.170**	1	15.478	0	15.303
1	1	**13.680**	1	13.960	0	21.314
2	1	**14.249**	1	16.556	1	17.346
3	1	**11.470**	1	13.650	1	13.941
4	1	**7.100**	1	7.707	1	19.358
5	1	**15.808**	1	24.743	1	16.210
6	1	9.729	1	**9.523**	1	11.010
7	1	21.589	1	**17.263**	0	21.553
8	1	2.750	1	**2.635**	1	2.786
9	0	13.12	1	**19.155**	0	1.758
10	1	9.480	1	**9.254**	1	9.806
11	1	**6.436**	1	7.541	0	7.691
12	1	**3.940**	1	4.135	1	4.133
13	1	22.810	1	23.024	1	**22.412**
14	1	**4.246**	1	4.673	1	4.471
15	1	**6.829**	1	8.221	0	6.125
16	1	**5.150**	1	7.550	1	6.555
17	1	**11.269**	1	14.575	0	10.661
18	1	**11.511**	0	0.357	0	2.541
19	1	**14.612**	0	13.625	0	18.188
20	1	**17.640**	0	6.285	0	9.628
21	1	**7.580**	1	9.492	1	16.853

22	1	4.350	1	**3.822**	1	3.943
23	1	15.689	1	**12.817**	1	15.867
24	1	3.315	1	**2.545**	1	2.550
25	1	**6.323**	0	6.779	1	7.641
26	1	**17.050**	1	17.992	1	22.714
27	1	8.592	1	**8.286**	0	2.259
28	1	**23.453**	0	19.811	1	29.246
29	1	13.080	1	**12.528**	0	0.851

Table A.9: Results of comparison (with 30 episodes) on point-goal driven exploration in Cantwell between classical algorithm, OccAnt (depth), and ANS (depth).

List of Figures

List of Tables