UNIVERSITY
OF BRESCIA

Integrating Planning and Learning for Agents Acting in Unknown Environments

Ph.D. in Information Engineering

Department of Information Engineering
University of Brescia
Data and Knowledge Management Unit
Fondazione Bruno Kessler

XXXV°cycle 2019–2022

Advisor:
Prof. Alfonso Emilio Gerevini

Co-Advisor:
Dr. Paolo Traverso

Ph.D. Candidate:
Leonardo Lamanna

Contents

Abstract

An Artificial Intelligence (AI) agent acting in an environment can perceive the environment through sensors and execute actions through actuators. Symbolic planning provides an agent with decision-making capabilities about the actions to execute for accomplishing tasks in the environment. For applying symbolic planning, an agent needs to know its symbolic state, and an abstract model of the environment dynamics. However, in the real world, an agent has low-level perceptions of the environment (e.g. its position given by a GPS sensor), rather than symbolic observations representing its current state. Furthermore, in many real-world scenarios, it is not feasible to provide an agent with a complete and correct model of the environment, e.g., when the environment is unknown a priori. The gap between the high-level representations, suitable for symbolic planning, and the low-level sensors and actuators, available in a real-world agent, can be bridged by integrating learning, planning, and acting.

Firstly, an agent has to map its continuous perceptions into its current symbolic state, e.g. by detecting the set of objects and their properties from an RGB image provided by an onboard camera. Afterward, the agent has to build a model of the environment by interacting with the environment and observing the effects of the executed actions. Finally, the agent has to plan on the learned environment model and execute the symbolic actions through its actuators.

We propose an architecture that integrates learning, planning, and acting. Our approach combines data-driven learning methods for building an environment model online with symbolic planning techniques for reasoning on the learned model. In particular, we focus on learning the environment model, from either continuous or symbolic observations, assuming the agent perceptual input is the complete and correct state of the environment, and the agent is able to execute symbolic actions in the environment. Afterward, we assume a partial model of the environment and the capability of mapping perceptions into noisy and incomplete symbolic states are given, and the agent has to exploit the environment model and its perception capabilities to perform tasks in unknown and partially observable environments. Then, we tackle the problem of online learning the mapping between continuous perceptions and symbolic states, assuming the agent is given a partial model of the environment and is able to execute symbolic actions in the real world.

In our approach, we take advantage of learning methods for overcoming some of the simplifying assumptions of symbolic planning, such as the full observability of the environment, or the need of having a correct environment model. Similarly, we take advantage of symbolic planning techniques to enable an agent to autonomously gather relevant information online, which is necessary for data-driven learning methods. We experimentally show the effectiveness of our approach in simulated and complex environments, outperforming state-of-the-art methods. Finally, we empirically demonstrate the applicability of our approach in real environments, by conducting experiments on a real robot.

Abstract

Un agente artificiale intelligente percepisce il proprio ambiente tramite sensori ed esegue azioni nell'ambiente tramite attuatori. La pianificazione simbolica permette agli agenti di decidere che azioni compiere al fine di eseguire delle attivita'. Per utilizzare la pianificazione simbolica, un agente artificiale deve conoscere il proprio stato simbolico e un modello astratto della dinamica dell'ambiente. Tuttavia, nel mondo reale, un agente ha delle percezioni di basso livello dell'ambiente (es. la propria posizione ritornata da un sensore GPS), piuttosto che osservazioni simboliche che descrivono il suo stato. Inoltre, in vari scenari reali, non e' possibile fornire a un agente un modello corretto e completo dell'ambiente, per esempio quando l'ambiente non e' noto a priori. Il divario tra rappresentazioni di alto livello, adatte per la pianificazione simbolica, e i sensori e attuatori di basso livello, disponibili in agenti artificiali reali, puo' essere colmato integrando apprendimento, pianificazione ed esecuzione.

Inizialmente, un agente deve mappare le percezioni continue in uno stato simbolico, per esempio riconoscendo gli oggetti e le relative proprieta' in un'immagine RGB fornita da una telecamera. Successivamente, l'agente deve costruire un modello dell'ambiente, interagendo con l'ambiente e osservando gli effetti delle proprie azioni. Infine, l'agente deve pianificare con il modello imparato, ed eseguire le azioni simboliche tramite i propri attuatori.

Proponiamo un'architettura the integra apprendimento, pianificazione ed esecuzione. Il nostro approccio combina metodi di apprendimento guidati dai dati, per costruire un modello dell'ambiente in tempo reale, con tecniche di pianificazione simbolica per ragionare sul modello imparato. In particolare, ci focalizziamo sull'apprendimento del modello dell'ambiente, sia da osservazioni continue che simboliche, assumendo che l'agente percepisca lo stato corretto e completo dell'ambiente, e che sia in grado di eseguire azioni simboliche nell'ambiente. Successivamente, assumiamo che siano dati un modello parziale dell'ambiente e la capacita' di mappare le percezioni in stati parzialmente corretti e incompleti, e l'agente deve sfruttare il modello dell'ambiente e la propria capacita' percettiva per assolvere dei compiti in ambienti sconosciuti a parzialmente osservabili. Inoltre, consideriamo il problema di imparare come mappare le percezioni continue in stati simbolici, assumendo che l'agente sia provvisto di un modello parziale dell'ambiente, e che sia in grado di eseguire azioni simboliche nel mondo reale.

Nel nostro approccio, sfruttiamo metodi di apprendimento per superare alcune assunzioni simplificative della pianificazione simbolica, come la completa osservabilita' dell'ambiente, o la necessita' di avere un modello corretto dell'ambiente. Analogamente, sfruttiamo tecniche di pianificazione simbolica per permettere a un agente di acquisire autonomamente i dati necessari per applicare i metodi di apprendimento. Mostriamo sperimentalmente l'efficacia del nostro approccio in ambienti simulati e complessi, ottenendo risultati migliori di metodi stato dell'arte. Infine, dimostriamo empiricamente l'applicabilita' del nostro approccio in ambienti reale, conducendo esperimenti su un robot reale.

Chapter 1

Introduction

An Artificial Intelligence (AI) agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [98]. For example, a robotic agent provided with an on-board RGB camera as a sensor, and wheel actuators for navigation. The general architecture of an AI agent is shown in Figure 1.1. An AI agent should be able to operate in unknown environments, e.g., a robotic agent designed for helping elderly people should be able to accomplish household tasks in different, a priori unknown, houses. When an agent is required to perform tasks in a known environment, it knows the actions that it can execute, and how the actions change the environment state. Therefore, it can plan the actions to execute for accomplishing tasks. However, when the environment is unknown, an agent has to learn how the environment works in order to make good decisions. The complexity of the agent environment depends on many factors (e.g. partially observable environments, multi-agent environments, dynamic environments, etc.). In particular, we focus on: *(i)* unknown environments, since the agent may have no knowledge about the action effects in the environment; *(ii)* continuous environments, given that perceptual input of the agent is provided by its continuous sensors (e.g. the agent position given by a GPS sensor); *(iii)* partially observable environments, where the agent has a partial view of the environment, e.g. the image provided by an on-board RGB camera; *(iv)* single-agent environment, since we do not deal with the cooperation or competition of multiple agents simultaneously operating in the same environment.

Enabling an agent to operate in unknown environments can be achieved by integrating learning, planning, and acting. On the learning side, an agent has to build and revise a model of the unknown environment where it finds itself in. Generally, a model of the environment can be obtained by combining commonsense input knowledge with the knowledge acquired by the agent while perceiving and interacting with the environment. The learned model should be suitable for reasoning. In particular, we focus on symbolic planning [39], which is a specific type of reasoning. Symbolic planning provides agents with decision-making capabilities about the actions to execute for achieving goals [98].

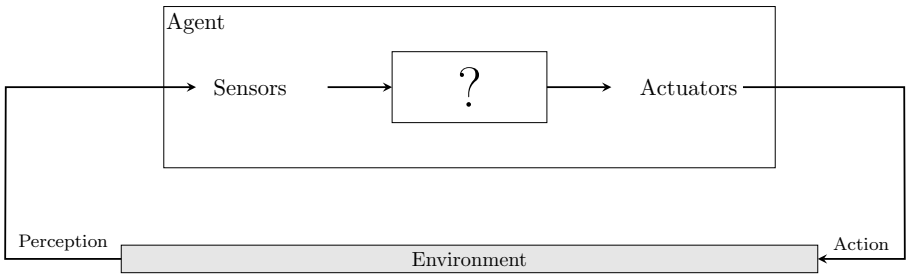


Figure 1.1: Agent perceiving the environment through sensors and executing actions in the environment through actuators. The agent behaviour is represented by the question mark box, which takes as input a perception and returns an action to execute.

Symbolic planning techniques are based on abstract and most often discrete representations of the world, where the agents perform their actions, usually called planning domains. A good planning domain should abstract away the details of the world state which are irrelevant to the achievement of the agent's goals. Finally, an agent need to be able to execute, through its actuators, the actions decided by means of symbolic planning.

We propose an agent architecture (Figure 1.2) that wraps up the learning, planning, and acting components, for an agent accomplishing tasks in an unknown environment. The agent executes low-level actions in the environment through its actuators, and perceives the environment through its sensors. We refer to the perceptual input of the agent as perception. The *perception function* maps a perception into symbols (i.e. objects and ground atoms) and anchors attributes to the objects (e.g. position, size, visual features, etc.). The output of the perception function is used to build and update the environment model, which is composed of a symbolic model of the environment, a symbolic state of the agent, and the anchors associated with the symbolic state objects. The symbolic model of the environment describes the environment dynamics, i.e., how the environment evolves when the agent executes actions. The symbolic model can be represented extensionally (e.g. by a finite state machine that describes the set of environment states and the transitions between states caused by actions) or intensionally by means of a planning language. A widely adopted planning language is the Planning Domain Definition Language (PDDL) [81]), which is a specification of the actions executable by the agent. In particular, in a PDDL model, each action is specified by its name, input parameters, preconditions, and effects. The preconditions are atoms that must be true (or false) in order to successfully execute the action. Similarly, the effects are atoms that become true (or false) after executing the action. The planner takes as input a planning problem composed of: a symbolic model of the environment, an agent symbolic state, and a goal specified by a first-order formula. The planner outputs a plan, if it exists, that is a solution to the planning problem. The executor takes as input the first symbolic action of the plan, returned by the

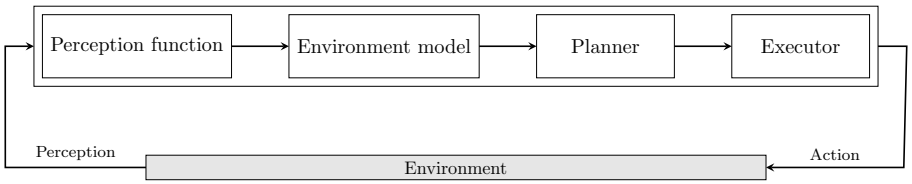


Figure 1.2: Agent environment interface.

planner module, and compiles the symbolic action into a sequence of low-level actions executable by the agent's actuators.

Learning the Perception Function A common assumption in symbolic planning is that the agent perceives the world at the symbolic level, e.g. it does not perceive its position through a GPS sensor, but directly perceives the fact that it is at a particular location, such as "my location is Rome". Due to this assumption, a fundamental problem arises when an agent wants to apply symbolic planning in a real-world scenario: how an agent can link the real-world (low-level) observations given by its sensors with the symbolic (high-level) observations used for applying symbolic planning. A possible resolution approach for the problem above consists of learning the mapping between continuous and symbolic observations. We refer to the function mapping perceptions into symbols as the *perception function*. In the AI literature, there is a research area studying this mapping between symbols and their continuous features, which is called *perceptual anchoring* [21]. In particular, *perceptual anchoring* is the process of creating and maintaining the correspondence between symbols and perceptions that refer to the same physical objects. The anchoring problem as defined in [21] applies a top-down approach: anchors are created starting from symbols and associating to symbols their perceptions. For example, the symbol "box_0" is associated with the anchor representing its image. The top-down perceptual anchoring can be seen as the symbol grounding problem restricted to physical objects. A different approach, i.e. bottom-up, has been applied in [61, 77], where they start from perceptions and link them to symbols. For example, a new anchor is created from an image and associated with a new random symbol. In this work, we aim to learn a *perception function* $\rho : X \to S$ where X is the perception space (e.g. the RGB images given by a camera) and S is the symbolic state space. For example, the perception space could be the set of RGB images given by a camera, and a symbolic state may be the set of objects detected in each image together with the predicates representing object properties (e.g. an object BOX_0 and the positive literal $\text{SMALL}(\text{BOX}_0)$).

An offline and modular approach for solving the problem of learning the perception function in complex environments (e.g. a kitchen where a robotic agent has to perform household tasks) is by training deep learning models that take as input RGB images and return the types and bounding boxes of the objects detected in the input RGB image. Afterward, other criteria can be used

for determining the properties of the detected objects, e.g. the estimation of the distance between the agent and an object can be used for determining whether the object is close to the agent. These approaches have the limitation that it is difficult to combine them with high-level commonsense knowledge, e.g. through neuro-symbolic reasoning that takes into account logical constraints about an object state.

An alternative approach for learning the perception function consists of end-to-end training of deep learning models that take as input the perceived state (e.g. the RGB image of the board of a game) and encode the perception into a vector of latent binary features. In particular, the latent binary features can be seen as symbolic variables describing a symbolic latent state. This approach has some limitations: *(i)* it requires perceptions of the fully observable state, since it fixes a priori the size of the state space, i.e. the length of the latent features vector; *(ii)* the learned representation of a state, albeit being symbolic, is not explainable, which can be a limitation when such a representation needs to be intelligible to humans (e.g. for communicating with humans). The two approaches proposed above are offline, since they require a dataset. A still open challenge is how to learn the perception function online, by interacting with the environment. For example, consider a deep learning model predicting the property isOpen for objects of type box. Instead of providing an agent with a pre-trained deep learning model, an agent should interact with a box (e.g. by opening/closing it) in order to learn what an open/closed box looks like, and consequently learn the perception function associated with the isOpen predicate online.

Learning to Act The problem of filling the gap between the symbolic level and the perception level is not only related to the construction of the agent's symbolic state. Indeed, when a robotic agent computes a symbolic plan, i.e. a sequence of symbolic actions, it cannot directly execute the symbolic actions through its actuators. For example, consider a robotic agent that can execute some navigation actions such as moving forward of a given number of centimeters or rotating of a given number of degrees. Suppose that the agent's goal is to be close to an object of type box, and that the symbolic plan consists of the single high-level action goCloseTo(box_0). The agent cannot directly execute the action goCloseTo(box_0), but it rather needs to compile it into a sequence of low-level navigation actions executable by its actuators. Similarly to the *perception function* ρ, we define an *action function* $\alpha : A \rightarrow C$, that associates to each high-level action in A a sequence of low-level actions in C. Learning the *action function* is very challenging when dealing with robotic agents that have to act in the real world. The majority of existing approaches tackle this problem by using Reinforcement Learning (RL) or hard-coded solutions (see Section 3.5). In this work, we do not focus on the problem of learning the *action function* online.

Learning the Environment Model Symbolic planning approaches require an agent to be provided with a symbolic model of the environment dynamics, describing how the environment state evolves when the agent executes actions. When specified in PDDL, such symbolic models are also referred to as PDDL action models. The manual specification of the action models is often an inaccurate, time-consuming, and error-prone task. The automated learning of action models is widely recognized as a key and compelling challenge to overcome these difficulties.

There is a wide variety of approaches (see Section 3.2) tackling the problem of learning action models offline, i.e. from an input set of plan traces. A plan trace is a trajectory in the state space generated by executing a plan, and is composed of the visited states and executed actions in the trajectory. These approaches have different assumptions on the correctness and observability of states and actions in the trajectory.

Few approaches focus on the problem of learning action models online, while executing actions. In the online setting, there is the additional complexity of generating the plan trace. The actions executed for generating a plan trace can be selected randomly, or by an oracle (e.g. a human). Alternatively, they can be selected by the planning itself, i.e., by solving a planning problem where the goal consists of learning a specific part of the action model (e.g. the effects of an action).

Example 1 (Planning, acting and learning agent)**.** *A robotic agent equipped with an RGB-D onboard camera and a position sensor is placed in an unknown kitchen and has to put an apple into a box. The agent perceives the RGB-D image of its egocentric view, and detects the objects in the image and their properties through its perception function, which can be a pre-trained deep learning model. The anchors of each object are their visual features extracted from a convolutional neural network that takes as input the RGB image cropped with the object bounding box, and its position which is estimated from the depth camera cropped with the object bounding box. The objects and their properties are used to update the symbolic state of the environment model. The object anchors are used to extend and update the set of anchors in the environment model. In this example, we assume the agent is provided with an input symbolic model of the environment, i.e. a planning domain specified in PDDL. After the environment model has been updated, the agent plans to achieve to goal of putting an apple into a box, then executes the actions in the plan until the goal is achieved. For example, if the first action of the plan is* $\textsc{goCloseTo}(\textsc{box}_0)$*, the agent uses a path planner for computing a path from its current position to the estimated position of* box_0 *in the topological map of the environment built online from the depth image while navigating into the environment. Finally, the path plan is translated into a sequence of low-level navigation operations such as moving forward of* $20cm$ *or rotating right of* 90 *degrees.*

The objective of this work is to provide a general architecture of a planning, learning, and acting agent, and propose solutions to some of the problems arising from the integration of planning, learning, and acting. Each problem

assumes that some planning, learning, and acting components are given and other components need to be learned. For example, the action model learning problem assumes that the agent is provided with a perfect perception function and executor components are given, i.e., the agent directly perceives the correct symbolic state of the environment, and is able to execute symbolic actions in the environment.

Specifically, we propose solutions to the following research questions:

- *How can an AI agent build an extensional representation of a planning domain from sensory data?*
- *How can an AI agent autonomously generate informative plan traces for learning an action model online?*
- *How can a robotic agent exploit an input action model for performing tasks in unknown and complex environments?*
- *How can a robotic agent reuse the previously acquired knowledge for solving a specific task in a particular environment?*
- *How can a robotic agent plan to learn the perception function for recognizing object properties?*

We evaluate the effectiveness of our approach for learning the environment model by comparing the learned models with the ground truth ones. Whereas the learned perception functions are evaluated by means of standard machine learning metrics, i.e. the precision and recall, computed on a test set of perceptions. Finally, the agent's capability of performing tasks is measured by taking into account whether the task is successfully executed, and the efficiency of its execution in terms of number of executed actions.

This work is organized as follows. Chapter 2 introduces some necessary background about symbolic planning, artificial neural networks, and RL; Chapter 3 deals with the related work. Chapter 4 addresses the problem of learning an extensional representation of a planning domain from sensory data. Chapter 5 describes a method for learning action models online in fully observable environments. Chapter 6 proposes a framework for agents that incrementally instantiate a symbolic planning domain, by planning, acting, and sensing, in an unknown environment. Chapter 7 addresses the challenge of planning for learning the perceptual capabilities of the agent. Chapter 8.2 tackles the problem of acquiring knowledge about unknown environments and reusing it to incrementally improve the agent performance. Finally, Chapter 9 gives conclusions and future directions of still open research challenges arising from the integration of planning, acting, and learning.

Chapter 2

Background

To apply symbolic planning, an agent needs to be provided with a planning problem, composed by: (i) a planning domain, which specifies the actions that can be executed by the agent; (ii) the agent's initial state; and (iii) a first-order goal formula representing the agent's goal. Generally, a goal formula identifies a set of goal states. The solution to the planning problem is a plan, i.e. a sequence of actions, that leads the agent from its initial state to a goal state. The language widely adopted to specify symbolic planning problems and domains is the Planning Domain Definition Language (PDDL) [81]. There are several kinds of planning (e.g. temporal planning, numerical planning, etc.). In this work, we focus on classical planning [39], where the agent's state is a set of boolean atoms, which become true or false after executing actions.

2.1 Classical Planning

Classical planning is a basic type of AI planning, where: *(i)* the agent state is represented by a set of propositional atoms (factored representation); *(ii)* states are fully observable, i.e. the truth value of all atoms is known in every agent state; *(iii)* the environment is deterministic, i.e. an action always has the same effects on the environment; *(iv)* the environment can change only due to actions executed by the agent.

Example 2. *Consider an agent capable to move blocks on a table. Suppose that there is a table* $table_0$ *with two blocks* $\{block_0, block_1\}$ *on top of* $table_0$*. The agent state can be represented as the set* s *of ground atoms* $\{\texttt{on}(block_0, table_0), \texttt{on}(block_1, table_0), \neg\texttt{on}(block_0, block_1), \neg\texttt{on}(block_1, block_0)\}$*, which is a full description of the environment state, i.e. the position of each block is fully observable. Whenever the agent stack* $block_1$ *on top of* $block_0$ *by executing the action* $\textsc{stack}(block_1, block_0)$ *in* s*, then the new state* s' *of the environment becomes* $\{\texttt{on}(block_1, block_0), \texttt{on}(block_0, table_0), \neg\texttt{on}(block_0, block_1), \neg\texttt{on}(block_1, table_0)\}$*, since the environment is deterministic. For example, it is assumed that* $block_1$ *cannot fall down while being stacked on top of* $block_0$*. Moreover, since the agent*

is the only one responsible for changes of the environment, it is assumed that no one other than the agent can move the blocks.

The PDDL specification of a classical planning problem is compliant with the so-called "database semantic assumption": *(i)* all literals that do not appear in the state description are negatives (aka closed-world assumption); *(ii)* each object (or constant) is uniquely identified by its name (aka unique name assumption).

Example 3. *In Example 2, according to the closed-world assumption, the state s can be specified by omitting the negative literals, i.e. $s = \{\texttt{on}(block_0, table_0), \texttt{on}(block_1, table_0)\}$. Moreover, for the unique name assumption, the name $table_0$ uniquely identifies the table, similarly $block_0$ and $block_1$.*

Let $\mathcal{P}$ be a set of first-order predicates, $\mathcal{O}$ a set of operators, $\mathcal{V}$ a set of variables (also called parameters), and $\mathcal{C}$ a set of constants. Predicates and operators of arity n are called n-ary predicates and n-ary operators. We use $\mathcal{P}(\mathcal{V})$ to denote the set of atoms $P(x_1, \ldots, x_m)$, where $x_i \in \mathcal{V}$ and $P \in \mathcal{P}$. For instance, if $\mathcal{P}$ contains the single binary predicate $\texttt{on}$, and $\mathcal{V} = \langle x_1, x_2, x_3 \rangle$. Then, $\mathcal{P}(\mathcal{V}) = \{\texttt{on}(x_i, x_j) \,|\, 1 \leq i, j \leq 3\}$. Similarly, we use $\mathcal{P}(\mathcal{C})$ to denote the set of atoms obtained by grounding $\mathcal{P}(\mathcal{V})$ with the constants in $\mathcal{C}$.

Definition 1 (Lifted action schema)**.** *A* lifted action schema *for an n-ary operator name $op \in \mathcal{O}$ on the set of predicates $\mathcal{P}$ is a tuple $\langle \mathsf{par}(op), \mathsf{pre}(op), \mathsf{eff}^+(op), \mathsf{eff}^-(op) \rangle$, where $\mathsf{par}(op) \subseteq \mathcal{V}$, $\mathsf{pre}(op)$, $\mathsf{eff}^+(op)$, and $\mathsf{eff}^-(op)$ are three sets of atoms on $\mathcal{P}(\mathsf{par}(op))$.*

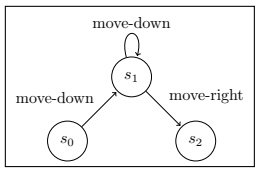
Essentially, $\mathsf{pre}(op)$, $\mathsf{eff}^+(op)$, and $\mathsf{eff}^-(op)$ represent the preconditions, positive, and negative effects of operator op. Without loss of generality, we assume that operators have no negative precondition.

Definition 2 (Ground action)**.** *The* ground action $a = op(c_1, \ldots, c_n)$ *of an n-ary operator name $op \in \mathcal{O}$ w.r.t. the constants $c_1, \ldots, c_n$ is the triple $\langle \mathsf{pre}(a), \mathsf{eff}^+(a), \mathsf{eff}^-(a) \rangle$, where $\mathsf{pre}(a)$ (resp. $\mathsf{eff}^+(a)$, $\mathsf{eff}^-(a)$) is obtained by replacing the i-th parameter of $\mathsf{par}(op)$ in $\mathsf{pre}(op)$ (resp. $\mathsf{eff}^+(op)$, $\mathsf{eff}^-(op)$) with c_i.*

We use the term *lifted*, as the opposite of *grounded*, to refer to expressions and actions where constants have been replaced with parameters.

Definition 3 (Planning domain)**.** *A planning domain $\mathcal{M}$ is a triple $\langle \mathcal{P}, \mathcal{O}, \mathcal{H} \rangle$ where $\mathcal{P}$ is a set of predicates, $\mathcal{O}$ is a set of operator names with their arity and, for every $op \in \mathcal{O}$, $\mathcal{H}$ is a function mapping an operator name op into a lifted action schema.*

Definition 4 (Finite-State Machine of a planning domain)**.** *The* Finite-State Machine (FSM) *of a planning domain $\mathcal{M} = \langle P, \mathcal{O}, \mathcal{H} \rangle$ for the set $\mathcal{C}$ of constants is the triple $\mathcal{M}(\mathcal{C}) = \langle \mathcal{S}, \mathcal{A}, \delta \rangle$ where $\mathcal{S} = 2^{\mathcal{P}(\mathcal{C})}$ is the set of all possible subsets of facts; $\mathcal{A}$ is the set of all possible ground actions of each n-ary operator name in $\mathcal{O}$ on any n-tuple of constants in $\mathcal{C}$; $\delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a transition relation such that $(s, a, s') \in \delta$ if $\mathsf{pre}(a) \subseteq s$ and $s' = s \cup \mathsf{eff}^+(a) \setminus \mathsf{eff}^-(a)$.*



(a) Extensional representation

```
move(roomX, roomY)
preconditions: (at-robby roomX)
                and (link roomX roomY)
effects: (at-robby roomY)
          and not (at-robby roomX)
```

(b) Intensional representation

Figure 2.1: Alternative representations of a planning domain.

Observation 1. *$\mathcal{M}$ is a* deterministic *planning domain if, given a set $\mathcal{C}$ of constants, the transition function δ of $\mathcal{M}(\mathcal{C})$ is deterministic, i.e., $\forall(s_{i-1}, a_i, s_i) \in \delta$, $\nexists(s_{i-1}, a_i, s'_i) \in \delta$ where $s_i \neq s'_i$.*

Observation 2. *$\mathcal{M}$ is a* discrete *planning domain if, given a set $\mathcal{C}$ of constants, the transition function δ of $\mathcal{M}(\mathcal{C})$ is discrete, i.e, the number of elements in the domain $\mathcal{S} \times \mathcal{A}$ is finite.*

Given a set of constants $\mathcal{C}$, a planning domain $\mathcal{M}$ can be represented explicitly by $\mathcal{M}(\mathcal{C})$, as in Figure 2.1a, or implicitly by $\mathcal{M}$ itself (Figure 2.1b). Indeed, $\mathcal{M}(\mathcal{C})$ can be obtained by instantiating $\mathcal{M}$ with $\mathcal{C}$, i.e., $\mathcal{A}$ can be obtained by grounding the lifted action schemas in $\mathcal{M}$ with $\mathcal{C}$, similarly $2^{\mathcal{P}(\mathcal{C})}$ can be obtained by grounding the predicates in $\mathcal{P}$ with $\mathcal{C}$. It is worth noting that $\mathcal{M}$ is a much more compact and general representation, since it does not require to explicitly enumerate all possible states and transitions of $\mathcal{M}(\mathcal{C})$, and it can be instantiated with different sets of constants.

Definition 5 (Planning problem)**.** *A* planning problem *is a tuple $\langle \mathcal{M}, \mathcal{C}, s_0, \mathcal{G} \rangle$ where $\mathcal{M}$ is an action model, $\mathcal{C}$ is a (possibly empty) set of constants, $s_0 \subseteq \mathcal{P}(\mathcal{C})$ is the initial state, and $\mathcal{G}$ is a first-order formula over $\mathcal{P}$, $\mathcal{V}$ and $\mathcal{C}$.*

Definition 6 (Plan)**.** *A* plan *for a planning problem $\langle \mathcal{M}, \mathcal{C}, s_0, \mathcal{G} \rangle$ is a sequence $\langle op_1(\boldsymbol{c}_1), \ldots, op_n(\boldsymbol{c}_n) \rangle$ such that there is a sequence $\langle s_1, \ldots, s_n \rangle$ of subsets of $\mathcal{P}(\mathcal{C})$ (aka states), such that for every $0 \leq i < n$, $\mathsf{pre}(op_i(\boldsymbol{c}_i)) \subseteq s_i$, $s_i = s_{i-1} \cup \mathsf{eff}^+(op_i(\boldsymbol{c}_i)) \setminus \mathsf{eff}^-(op_i(\boldsymbol{c}_i))$, and $s_n \models \mathcal{G}$.*

A state $s_n \in S$ is *reachable* from a state $s_0 \in S$ in $\mathcal{M}(\mathcal{C})$ if there is a plan $\langle a_1, \ldots, a_n \rangle$ such that $(s_{i-1}, a_i, s_i) \in \delta$ for $i = 1 \ldots n$.

Notice that our definition of planning problem allows to express the first-order goal formula $\mathcal{G}$. We say that a state $s \models \mathcal{G}$ iff $\bigwedge_{P(\boldsymbol{c}) \in s} P(\boldsymbol{c}) \wedge \bigwedge_{P(\boldsymbol{c}) \in \mathcal{P}(\mathcal{C}) \setminus s} \neg P(\boldsymbol{c}) \models \mathcal{G}$, under the assumption that all the elements of the problem are in $\mathcal{C}$.

2.1.1 Solving Planning Problems

State space search There are several approaches for solving a planning problem: state space search, plan space search, boolean satisfiability, situation calculus, etc. In the following, we describe state space search approaches, since they are the ones mainly used in this work.

A search algorithm is said to be *complete* when it is guaranteed to find all possible solutions, and *sound* if it always computes correct solutions. The search in the state space can be forward or backward. The forward search starts from an initial state, and searches for a plan that leads the agent from the initial state to a goal state. This can be achieved by considering all possible actions applicable from the current state (e.g. breadth search), and, in this case, the search algorithm is both complete and sound. However, performing a sound and complete forward search is typically unfeasible, since the state space of a planning problem is exponentially large w.r.t. the ground atoms of the planning domain states (i.e. with n ground atoms the size of the state space is 2^n).

The backward state space search starts from a set of goal states, and looks for a backward sequence of actions that leads from a goal state to the initial state. A key difference between the backward and forward search, is that in the backward case the search algorithm considers a set of states, rather than a single state. Moreover, at each step, the forward search considers the applicable actions in a state, and the backward search considers the *relevant* actions for a goal. A ground action is *relevant* for a goal (represented as a conjunction of literals), if at least one of the action's positive/negative effects belongs to the goal. The action considered by the backward search should be the last action of the plan, while in the forward search it should be the first action of the plan.

Heuristic functions It is worth noting that both forward and backward searches suffer from scalability drawbacks. For this reason, typically heuristic functions are exploited. In particular, a heuristic function estimates, for each state s, the cost for reaching a goal state starting from s.

A well-known example of a (best-first) search algorithm, which makes use of a heuristic function, is the A^* search [98]. Specifically, A^* exploits a cost function f for assigning to each state s a cost $f(s) = g(s) + h(s)$ where $g(s)$ is the cost for reaching the state s from the initial state, and $h(s)$ is the heuristic function estimating the cost of the cheapest path from s to a goal state. For further details about the theoretical properties of a heuristic function and the A^* search algorithm we refer to [98].

In the state space search, there are two main approaches for designing a heuristic function: *(i)* adding transitions to the FSM of the planning domain; *(ii)* removing states from the FSM of the planning domain. An example of a heuristic that adds transitions is the "ignore preconditions heuristic", where the original planning problem is relaxed by removing all action preconditions, and all action effects but goal conditions. Next, the estimated cost for achieving the goal from a state s is the length of the plan solving the relaxed planning problem from s. A simple example of a heuristic that removes states from the FSM of the planning domain is a heuristic that removes some ground atoms from the initial state. This approach is also known as *state abstraction*, since many states of the original planning problem are mapped into a single, more abstract, state of the relaxed problem.

op	$\mathsf{par}(op)$	$\mathsf{pre}(op)$	$\mathsf{eff}^+(op)$	$\mathsf{eff}^-(op)$
PUTONTABLE	x_1, x_2	$\texttt{clear}(x_1), \texttt{on}(x_1, x_2),$ $\texttt{block}(x_1), \texttt{block}(x_2)$	$\texttt{clear}(x_2)$	$\texttt{on\_table}(x_1)$
PUTONBLOCK	x_1, x_2	$\texttt{clear}(x_1), \texttt{clear}(x_2),$ $\texttt{block}(x_1), \texttt{block}(x_2)$	$\texttt{on}(x_1, x_2)$	$\texttt{clear}(x_2)$

Table 2.1: Action schema of the operators in the blocksworld domain.

Figure 2.2: An example of planning problem in the blocksworld domain. The plan consists of the single action PUTONBLOCK(block1, block0).

2.1.2 Domain Examples

Blocksworld An agent has to move some blocks placed on a table. The agent state is described by the set of predicates $\mathcal{P} = \{\texttt{block}, \texttt{on}, \texttt{on\_table}, \texttt{clear}\}$. The unary predicate `block` describes the type of the input object, e.g. $\texttt{block}(block_0)$ indicates that $block_0$ is an object of type `block`. The unary predicate `on_table` takes as parameter an object of type `block` and indicates that the block is placed on the table. Similarly, the binary predicate `on` takes as parameters two objects of type `block` and indicates that the first block is placed on top of the second block.

The set of operator names is $\mathcal{O} = \{\text{PUTONBLOCK}, \text{PUTONTABLE}\}$. The operator PUTONBLOCK takes as parameters two objects of type `block` and places the first block on top of the second one. Similarly, PUTONTABLE takes as parameter an object of type `block` and places it on the table. The action schema of the operators in the blocksworld domain is reported in Table 2.1. An example of planning problem is reported in Figure 2.3.

Gripper An agent has to move some balls among different rooms of a building. The agent state is described by the set of predicates $\mathcal{P} = \{\texttt{room}, \texttt{ball}, \texttt{at\_robby}, \texttt{at\_ball}, \texttt{free}, \texttt{carry}\}$. The nullary predicate `free` indicates that the agent gripper is free, i.e., the agent is not carrying any ball. The unary predicate `carry` takes as parameter an object of type `ball` and indicates that the agent is carrying the ball. The unary predicates `room` and `ball` describe the type of the input object, e.g. $\texttt{room}(room_0)$ indicates that $room_0$ is an object of type `room`. The unary predicate `at_robby` takes as parameter an object of type `room` and indicates that the agent is located at the input room object. Similarly, the binary predicate `at_ball` takes

op	$\mathsf{par}(op)$	$\mathsf{pre}(op)$	$\mathsf{eff}^+(op)$	$\mathsf{eff}^-(op)$
MOVE	x_1, x_2	$\texttt{room}(x_1), \texttt{room}(x_2),$ $\texttt{at\_robby}(x_1)$	$\texttt{at\_robby}(x_2)$	$\texttt{at\_robby}(x_1)$
PICK	x_1, x_2	$\texttt{ball}(x_1), \texttt{room}(x_2),$ $\texttt{at\_robby}(x_2), \texttt{free}(),$ $\texttt{at\_ball}(x_1, x_2)$	$\texttt{carry}(x_1)$	$\texttt{at\_ball}(x_1, x_2),$ $\texttt{free}()$
DROP	x_1, x_2	$\texttt{ball}(x_1), \texttt{room}(x_2),$ $\texttt{at\_robby}(x_2), \texttt{carry}(x_1)$	$\texttt{at\_ball}(x_1, x_2),$ $\texttt{free}()$	$\texttt{carry}(x_1)$

Table 2.2: Action schema of the operators in the blocksworld domain.

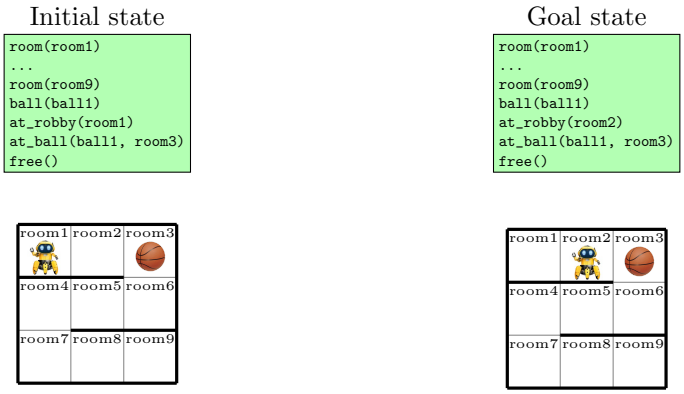


Figure 2.3: An example of planning problem in the gripper domain. The plan consists of the single action MOVE(room1, room2).

as parameter an object of type `ball` and an object of type `room`, and indicates that the ball object is located at the room object.

The set of operator names is $\mathcal{O} = \{\text{MOVE}, \text{PICK}, \text{DROP}\}$. The operator MOVE takes as input parameters two objects of type `room` and moves the agent from the first room to the second one. The operator PICK takes as parameter two objects of type `ball` and `room` and means that the agent picks up the ball placed in the room where it is located into. Similarly, the operator DROP takes as parameter an object of type `ball` and an object of type `room` and indicates that the agent drops the ball into the room where it is located into. The action schema of the operators in the gripper domain are reported in Table 2.2. An example of planning problem is reported in Figure 2.2.

2.2 Supervised Learning

An agent is learning if it improves its performance on future tasks after making observations about the world [98]. A commonly adopted taxonomy of learning groups it into three main paradigms: supervised learning, unsupervised learning,

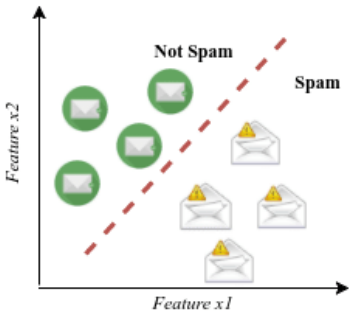


Figure 2.4: Classification

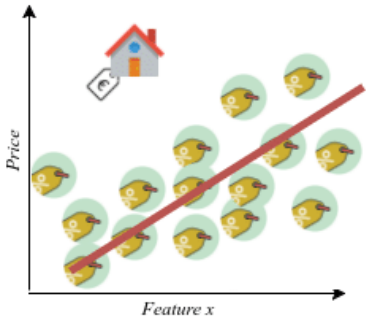


Figure 2.5: Regression

and reinforcement learning.

Supervised learning [25] consists of learning an approximation of a function $f : X \to Y$ given a set of n pairs $\langle x_1, y_1 \rangle$, $\langle x_2, y_2 \rangle$, ..., $\langle x_n, y_n \rangle$, namely a training set, where $x_i \in X$ and $y_i \in Y$, for $i \in 1, \ldots, n$. X is referred to as the input set and is composed of real value numbers, similarly Y is the output set, which can be either discrete or continuous. When Y is a set of discrete values (aka classes), f is a *classification* function, and learning f allows to solve a *classification* problem (e.g. binary classification [67]). Similarly, when Y is a set of continuous values, f is a *regression* function, which solves a *regression* problem (e.g. linear regression [80]).

Typically, learning f is not feasible since the training set does not contain all the possible pairs $\langle x_i, y_i \rangle$ that can be obtained by considering every element in X, and the corresponding output element in Y. Therefore, the learned function, denoted as f' and referred to as an *hypothesis*, is an approximation of f. The function f' should generalize over unseen input elements, i.e., given a new input element $x_{new} \in X$, $f'(x_{new})$ should be equal to $f(x_{new})$.

The learned hypothesis is evaluated by means of a test set, which consists of m pairs $\langle x_1, y_1 \rangle$, $\langle x_2, y_2 \rangle$, ..., $\langle x_m, y_m \rangle$. Typically, $m < n$, and the test and training sets are disjoint. It is worth noting that evaluating the agent on pairs $\langle x, y \rangle$ contained in the training set is likely to overestimate the generalization performance of f' on unseen examples. This problem is well-known as *overfitting*, i.e. the learned function f' provides good accuracy on the training set but worse accuracy on the test set. This is because the learned f' takes too much care of details in the training set examples, which are not relevant to the class prediction. As a result, the performance of f' on the test set worsens, i.e. f' does not generalize well on unseen examples.

2.2.1 Artificial Neural Networks

Artificial neural networks (or shortly neural networks in the following) are computational models inspired by the human brain: they simulate the mental activity under the (neuroscience) hypothesis that it consists primarily of electrochem-

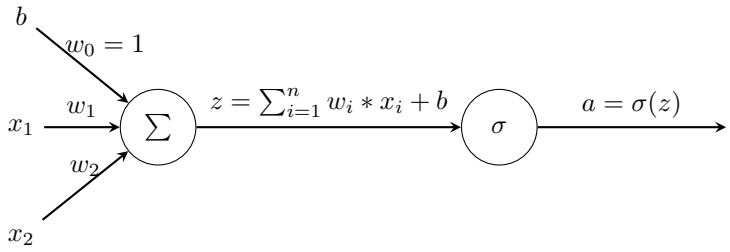


Figure 2.6: A perceptron.

ical activity in networks of brain cells [98], called neurons. A neural network is composed of units (also called perceptrons), the model of a single unit is shown in Figure 2.6. The unit takes as input a bias $b \in \mathbb{R}$, and an n-dimensional vector of data $\mathbf{x} = \{x_1, \ldots, x_n\} \in \mathbb{R}^n$, which is associated with an n-dimensional vector of weights $\boldsymbol{w} = \{ w_0, \ldots, w_{n-1}\} \in \mathbb{R}^n$. The weights in $\boldsymbol{w}$ and the bias b are commonly referred to as parameters of the neural network. The unit computes the sum of $\boldsymbol{w} \cdot \mathbf{x} + \boldsymbol{b}$ and applies an activation function a to derive the output. An example of an activation function for binary classification is the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid function takes as input a real value x and outputs a value $\sigma(x) \in (0, 1)$, which can be seen as a truth probability. In particular, the larger the input value x, the closer the output value $\sigma(x)$ to 1; similarly, the smaller the input x, the closer the output $\sigma(x)$ to 0, as shown in Figure 2.7.

Another example of an activation function is the softmax function S, which is a generalization of the sigmoid function that can be used for multi-class classification:

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The softmax function maps an input vector of n real values to an output vector of n real values that sum to 1. The *i-th* value of the output vector corresponds to the probability of the input vector belonging to the *i-th* class.

There exist many activation functions, for a comprehensive survey about different activation functions we refer to [31].

Feed-forward Neural Networks A neural network is generally composed of many units connected with each other and grouped in layers. In a feed-forward neural network, units of subsequent layers are fully connected, i.e., the output of each unit in the previous layer is given as input to each unit in the next layer. In particular, when the input layer is directly connected to the output layer, the network is a single-layer feed-forward neural network. Whereas in the

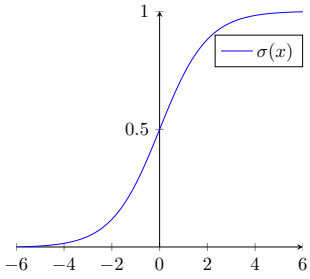


Figure 2.7: The sigmoid activation function.

architecture of a multilayer feed-forward neural network there is at least one hidden layer. An example of a multilayer feed-forward neural network with one hidden layer is shown in Figure 2.8.

Universal approximation theorem Notably, it has been proved that multilayer feed-forward neural networks with a single hidden layer can approximate any continuous function to any desired precision [52]. However, even if such networks are able to approximate any continuous function, learning the approximation function may fail for two main reasons: *(i)* the optimization algorithm used for learning the parameters of the network may not be able to find the parameter values corresponding to the desired approximation function; *(ii)* the learned approximation function might be not correct due to overfitting. Feed-forward networks provide a universal system for representing functions in the sense that, given a function, there exists a feed-forward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to examples not in the training set [42].

Loss function The parameters of a neural network are optimized for minimizing the prediction error, which is computed by means of a loss (or cost) function. Given a labeled input pair $\langle x, y \rangle$, the loss function $\mathcal{L}(y, \hat{y})$ compares the network prediction $f(x, \theta) = \hat{y}$ with the target value y. An example of a loss function typically adopted for binary classification is the binary cross-entropy [95]. Given a set of labeled input pairs $\{\langle x_i, y_i \rangle\}_{i=1}^{N}$, the binary cross-entropy can be defined as:

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)$$

In particular, for each positive input pair $\langle x_i, y_i = 1 \rangle$ the binary cross-entropy sums $log(\hat{y})$, which is the log probability of x_i being classified as 1. Indeed, if the probability associated with the true class is close to 1, then its contribution to the loss is close to zero. Conversely, if that probability is low,

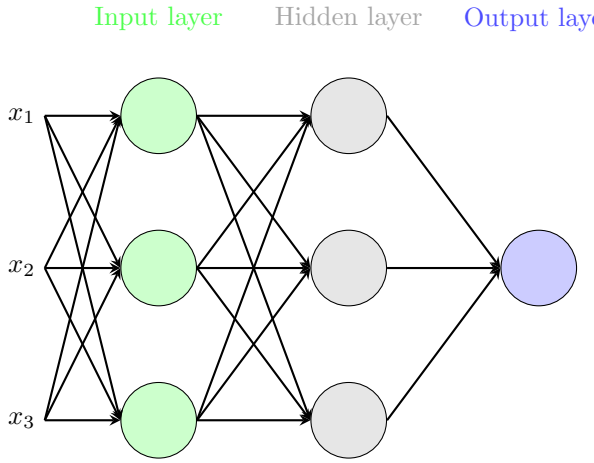


Figure 2.8: A multilayer feed-forward neural network with one hidden layer.

then the contribution to the loss is high. Similarly, for each negative input pair $\langle x_i, y_i = 0 \rangle$, the binary cross-entropy sums $log(1 - \hat{y})$.

There exist several loss functions [115], and the choice of the loss function strictly depends on the task accomplished by the neural network (e.g. cross entropy for multi-class classification tasks, or mean squared error for regression tasks).

Back-propagation To make neural network predictions more accurate, we have to change the network parameters (i.e. the weights and the bias). This can be achieved by applying the gradient descent algorithm. In particular, each parameter w can be updated as:

$$w \leftarrow w - \mu \frac{\partial \mathcal{L}}{\partial w}$$

where the learning rate μ is a value in the range $[0, 1]$, and $\frac{\partial \mathcal{L}}{\partial w}$ is the derivative of the loss with respect to the parameter w. The derivative $\frac{\partial \mathcal{L}}{\partial w}$ can be computed by applying the chain rule [117].

For example, consider two subsequent layers with one node for each layer, in particular, the output layer and the previous hidden layer, as shown in Figure 2.9. Suppose that a is the sigmoid activation function. The input of the output node is $a^{(L)} = \sigma(z^{(L)})$, where $z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$. Consider the loss function $\mathcal{L}_0 = (\hat{y}_0 - y_0)^2$, where $\hat{y}_0 = a^{(L)}$. Since $\mathcal{L}$ depends on $a^{(L)}$, and $a^{(L)}$ depends on $z^{(L)}$, and $z^{(L)}$ depends on $w^{(L)}$, to compute the derivative $\frac{\partial \mathcal{L}_0}{\partial w^{(L)}}$, we apply the chain rule and obtain:

$$\frac{\partial \mathcal{L}_0}{\partial w^{(L)}} = \frac{\partial \mathcal{L}_0}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

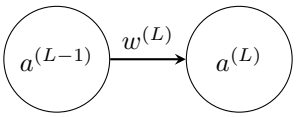


Figure 2.9: Link between two nodes of subsequent layers.

Specifically, in our example we obtain:

$$\frac{\partial \mathcal{L}_0}{\partial a^{(L)}} = 2(a^{(L)} - y_0)$$
$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$
$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Therefore:

$$\frac{\partial \mathcal{L}_0}{\partial w^{(L)}} = 2(a^{(L)} - y_0)\sigma'(z^{(L)})a^{(L-1)}$$

It is worth noting that the derivative $\frac{\partial \mathcal{L}_0}{\partial w^{(L)}}$ above is defined for a single training example, though it can be generalized to a set of N training examples by averaging over all training examples, i.e.

$$\frac{\partial \mathcal{L}}{\partial w^{(L)}} = \frac{1}{N}\sum_{k=1}^{N}\frac{\partial \mathcal{L}_k}{\partial w^{(L)}}$$

Similarly, the bias $b^{(L)}$ can be updated by computing the derivative of the loss with respect to the bias:

$$\frac{\partial \mathcal{L}_0}{\partial b^{(L)}} = \frac{\partial \mathcal{L}_0}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial b^{(L)}} = 2(a^{(L)} - y_0)\sigma'(z^{(L)})$$

The above procedure can be generally applied to compute the derivative with respect to the weight in the previous layer:

$$\frac{\partial \mathcal{L}_0}{\partial w^{(L-1)}} = \frac{\partial \mathcal{L}_0}{\partial a^{(L-1)}}\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}$$

However, since the output of the perceptron in the layer $L-1$, i.e. $a^{(L-1)}$, is the input of the perceptron in the layer L, the output error for the perceptron in layer $L-1$ is the input error of the perceptron in layer L, i.e. $\frac{\partial \mathcal{L}_0}{\partial a^{(L-1)}}$. Similarly to $\frac{\partial \mathcal{L}_0}{\partial w^{(L)}}$ and $\frac{\partial \mathcal{L}_0}{\partial b^{(L)}}$, the derivative $\frac{\partial \mathcal{L}_0}{\partial a^{(L-1)}}$ can be defined as:

$$\frac{\partial \mathcal{L}_0}{\partial a^{(L-1)}} = \frac{\partial \mathcal{L}_0}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = 2(a^{(L)} - y_0)\sigma'(z^{(L)})w^{(L)}$$

When an output layer L is composed of n_L units (Figure 2.10), each *j-th* output can be denoted as $a_j^{(L)}$ and the loss $\mathcal{L}_0$ is computed by summing over all

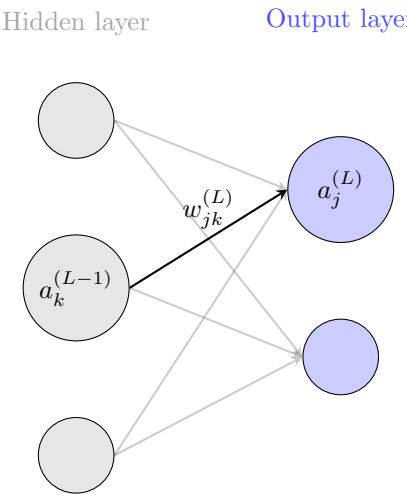


Figure 2.10: The hidden layer and output layer of a multilayer feed-forward neural network with multiple units for each layer.

output units, i.e., $\mathcal{L}_0 = \sum_{j=0}^{n_L-1}(a_j^{(L)} - y_{0,j})^2$, where $y_{0,j}$ is the *j-th* component of the label y_0. Similarly, $a_j^{(L)} = \sigma(z_j^{(L)})$, where the input $z_j^{(L)}$ of the *j-th* unit in the *L-th* layer is:

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)}$$

Summing up, the derivative of the loss $\mathcal{L}_0$ with respect to the weight $w_{jk}^{(L)}$ is:

$$\frac{\partial \mathcal{L}_0}{\partial w_{jk}^{(L)}} = \frac{\partial \mathcal{L}_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$$

Finally, the derivative of the loss with respect to one activation $a_k^{(L-1)}$ of the previous layer $L-1$ becomes:

$$\frac{\partial \mathcal{L}_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial \mathcal{L}_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} =$$

It is worth noting that each unit in the layer $L-1$ affects the loss of each unit in the layer L, since the two layers are fully connected, therefore $\frac{\partial \mathcal{L}_0}{\partial a_k^{(L-1)}}$ sums over all units in the layer L.

Optimizers The parameters of a neural network are updated for reducing the loss according to an optimization algorithm, commonly referred to as an optimizer. An example of an optimizer is the gradient descent algorithm previously

considered. Generally, there exist several optimizers, each with its strength and weaknesses, for a detailed overview we refer to [96].

Hyperparameters The hyperparameters of a neural network are variables that determine the network structure (e.g. number of layers and units) and the training process (e.g. learning rate, weights initialization method, number of training epochs, etc.). The hyperparameter values are set before starting the model training process, i.e. before optimizing the parameters. There exist several approaches for optimizing the hyperparameters: random/grid search, Bayesian optimization, etc. For a detailed survey about hyperparameter optimization algorithms we refer to [123].

2.3 Reinforcement Learning

Reinforcement Learning (RL) is based on the idea that an agent learns by perceiving and interacting with its environment [110]. For example, consider an agent that is learning to play chess. The agent can learn a transition model by making different moves. However, without external feedback about what is (or is not) a desirable state, the agent has no grounds for deciding which move to make. The agent needs to be able to understand that a good state is when it can checkmate the opponent, and that when it is checkmated it is in a bad state. The feedback discriminating good or bad states is called reward, and strictly depends on the goal of the agent related to the state of the environment. RL is learning how to map states to actions, so as to maximize a numerical reward signal. An RL agent has explicit goals, can sense aspects of its environment, and can choose actions to influence the environment. Formalizing the idea of a goal through a reward signal is one of the most distinctive features of RL. The agent environment interface of an RL agent is shown in Figure 2.11. At each time step t, the agent executes an action a_t, and, at the next time step $t+1$, it ends up in a state s_{t+1} with a reward r_{t+1}.

Exploitation VS exploration For maximizing the reward, the agent has to execute actions previously tried that produced a high reward. However, to discover good actions in terms of achieved reward, the agent has to try to execute different new actions. This trade-off is known as the exploration-exploitation dilemma, the agent has to exploit what it has experienced in the past, but it also has to explore in order to make better action selections in the future.

RL basic elements Beyond the agent and the environment, there are four main components of an RL system:

– a *policy* function $\pi : S \to A$, where S is the set of perceived environment states and A is the set of actions executable by the agent. The policy function defines the agent's behavior, by a mapping from perceived states to actions to be executed in those states. In general, a policy can be

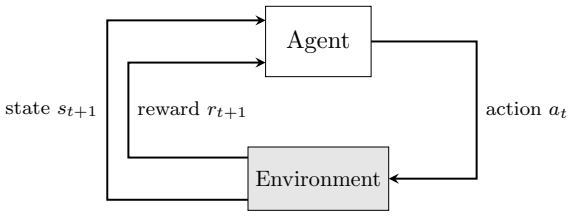


Figure 2.11: A Reinforcement Learning agent environment interface. The agent executes actions in the environment and perceives the environment state and a reward.

stochastic, specifying probabilities of the actions executable in a perceived state.

- The *reward* function defines the goal of an RL problem. The agent's objective is to maximize the total reward it receives over the long run. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by a low reward, then the policy may be changed to select some other action in that situation in the future. Generally, reward signals may be stochastic functions of the state of the environment and the actions taken. The reward function can be formally defined as: $\mathcal{R} : S \times A \times S \to \mathbb{R}$

- Whereas the reward signal indicates what is good in the short-term, a *value function* specifies what is good in the long-term. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards, or vice versa.

  Without rewards there could be no values, and the only purpose of estimating values is to achieve more rewards. Nevertheless, decision-making is based on values: we prefer actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. Notice that rewards are basically directed by the environment, but values must be estimated and re-estimated from the sequences of the agent's observations. In fact, the most important component of many RL algorithms is a method for efficiently estimating values.

- The final element of some (i.e. model-based) RL systems is a *model* of the environment, which allows to make inference about how the environment behaves when the agent executes actions. Models are used for planning, by considering possible future situations before they are actually experienced. The model-based methods use models and planning for solving RL, as

opposed to simpler model-free methods that are explicitly trial-and-error viewed as almost the opposite of planning.

Markov Decision Processes Markov Decision Processes (MDPs) are a classical formalization of sequential decision-making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to trade off immediate and delayed reward.

Formally, an MDP is a tuple $\langle S, A, R, P, \gamma \rangle$, where S is a set of environment states, A is a set of actions executable by the agent, R is a reward function, and P is a stochastic transition function, i.e. $P(s, a, s') = Pr(s'|s, a)$. An MDP is *finite* when S and A are finite sets, otherwise the MDP is *infinite*.

It is worth noting that an MDP satisfies the *Markov property*, i.e., the distribution of the next state s' depends only on the current state s and action a, rather than on the whole sequence of previously visited states and executed actions; formally:

$$Pr(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) = Pr(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t)$$

Episodic VS continuous tasks The interaction between the agent and the environment can naturally break into finite subsequences called episodes, such as plays of a game. At each episode, the agent starts from a (possibly sampled) initial state and ends in a particular type of state named terminal state, which can be reached from the initial state by executing a finite sequence of actions. Interactions of this kind, i.e., where there is a natural notion of final time step, are called episodic tasks, and the final time step T is a random variable that can vary between different episodes. Similarly, when the agent–environment interaction goes on continually with no time step limit, such as an application to a robot with a long life span, the task is defined continuing task.

The time step limit T is also referred to as horizon. For episodic tasks, T is a finite number and the MDP has a *finite horizon*. Similarly, for continuing tasks, T is infinite and we have an *infinite horizon* MDP.

Return and discount At each time step t, the immediate reward is a single number, $r_t \in \mathbb{R}$. However, the agent does not have to maximize the immediate reward, but the cumulative reward in the long run. Let $r_{t+1}, r_{t+2}, \ldots$ be the sequence of rewards received by the agent after time step t. Formally, the agent seek to maximize the expected return G_t, which is a function of the reward sequence, e.g. $G_t = r_{t+1} + r_{t+2} + \cdots + r_T$.

Notice that the return formulation can be problematic for continuing tasks, because the final time step is not finite and the maximized expected return can be infinite (e.g. suppose the agent receives a reward of $+1$ at each time step). To maximize the expected return with infinite horizon, we introduce the discounted return. The agent tries to select actions so that the sum of the

discounted rewards it receives over the future is maximized. In particular, it chooses the next action to execute in order to maximize the expected discounted return:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \in [0, 1]$ is called the discount factor, and determines how much future reward should be "discounted" when making decisions. The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the expected discounted reward is a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is "myopic" in being concerned only with maximizing immediate rewards: its objective, in this case, is to learn how to choose A_t so as to maximize only r_{t+1}. In general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Finally, the definition of the return in the continuing and episodic tasks can be unified as follows:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

where $T = \infty$ for continuing tasks and it has to be $\gamma < 1$; whereas T is a finite value for episodic tasks and we admit the case $\gamma = 1$.

Value functions and policies Most RL algorithms involve estimating value functions, i.e. functions of states (or state action pairs) that estimate how good it is for the agent to be in a given state (or to perform a given action in a given state). Specifically, the notion of expected discounted reward allows to measure "how good" is a state in terms of future rewards. As future rewards depend on what actions the agent will take, value functions are defined with respect to particular ways of acting, called policies. Formally, a policy π is a mapping from states to probabilities of selecting each possible action, i.e., π defines a probability distribution over $a \in A(s)$ for each $s \in S$. If the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $a_t = a$ if $s_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s\right], \text{ for all } s \in \mathcal{S}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value* function

for policy π. Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s, A_t = a\right]$$

We call q_π the *action-value* function for policy π.

Bellman equations A fundamental property of value functions used throughout RL and dynamic programming is that they satisfy recursive relationships, also known as *Bellman consistency equations*. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r Pr(s', r|s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} Pr(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big], \quad \text{for all } s \in S, \end{aligned}$$

For each triple (a, s', r), we compute the probability of executing action $a \in A$ from state $s \in S$ and ending in state $s' \in S$ with reward $r \in R$, i.e., $\pi(a|s)p(s', r|s, a)$. Then, the probability associated with each triple (a, s', r) is used for weighting the discounted value $\gamma v_\pi(s')$ of the expected next state s', plus the reward r expected along the way. Finally, we sum over all possible triples to get an expected value. The Bellman consistency equation for v_π states that the value of the start state must equal the discounted value of the expected next state, plus the reward expected along the way.

Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π', for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$, for all $s \in S$. For finite MDPs, there always exists a policy π_* that is better than or equal to all other policies, i.e., an optimal policy. Generally, there can be multiple optimal policies. They share the same state–value function v_*, called the optimal state–value function and defined as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{for all } s \in \mathcal{S}$$

Optimal policies also share the same optimal action–value function q_*, defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state-action pair (s, a), the function $q_*(s, a)$ gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = s, a_t = a]$$

Given that v_* is the value function for a policy, it must satisfy the Bellman consistency equation previously stated. However, since v_* is the optimal value function, the Bellman consistency equation for v_* can be stated with no reference to any specific policy. This is the *Bellman optimality equation* for v_*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \max_a \mathbb{E}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma v_*(s')] \end{aligned}$$

The last two equations are two forms of the Bellman optimality equation for v_*; whereas the Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a') | s_t = s, a_t = a] \\ &= \sum_{s', r} p(s', r | s, a) \Big[r + \gamma \max_a{}' q_*(s', a') \Big] \end{aligned}$$

For finite MDPs, the Bellman optimality equation for v_* has a unique solution. The Bellman optimality equation is actually a system of n equations, one for each state. In principle, when the dynamics P of the environment is known, the nonlinear system of equations for v_* can be solved by means of state-of-the-art solvers of nonlinear systems; similarly for q_*.

Once the optimal value function v_* has been computed, then any policy that is greedy with respect to v_* is an optimal policy, i.e., the best actions after a one-step search will be optimal actions. A greedy policy is actually optimal in the long-term sense because v_* turns the optimal expected long-term return into a quantity that is locally and immediately available for each state.

Similarly with q_*, for any state s, the agent can select an action that maximizes $q_*(s, a)$. The action-value function provides the optimal expected long-term return as a value that is locally and immediately available for each state-action pair.

Explicitly solving the Bellman optimality equation allows to find an optimal policy, and thus to solve the RL problem. However, this solution is rarely applicable, since it relies on at least three strong assumptions: *(i)* a complete and correct MDP of the environment is known; *(ii)* there are enough computational resources to compute a solution; *(iii)* the Markov property. Typically, it is not possible to apply this solution since at least one of the above assumptions is violated. For example, when the problem has an exponential number of states, it is not feasible to compute a solution to a nonlinear system of an exponentially large number of equations in a reasonable time. Due to the aforementioned limitations, in RL it is typically necessary to look for approximate solutions, e.g., using actually experienced transitions in place of knowledge of the expected transitions.

Chapter 3

Related work

We refer to Figure 1.2, and describe the related work according to their simplifying assumptions with respect to the agent components to be learned. Many approaches focus on learning a specific component, e.g. perceptual anchoring approaches aim to learn the perception function, and typically do not consider agents planning in unknown environments; action model learning approaches assumes a perfect perception function and executor module are given and focus on learning the action preconditions and effects, i.e. the environment model. Different methods have been proposed for planning in a latent space, these methods do not consider the problem of symbolic action execution but tackle the problem of learning multiple components, i.e. the perception function and the environment model. Approaches integrating symbolic planning and deep RL assume all the components to be given but the executor, and exploit deep RL techniques for learning to compile symbolic actions into low-level operations executable by the agent's actuators. Finally, deep RL methods do not consider the agent components separately, but rather apply end-to-end training for learning all the components simultaneously.

3.1 Perceptual Anchoring

Perceptual anchoring [21] is the process of creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects. With respect to Figure 1.2, perceptual anchoring deal with the problem of learning the perception function and " Environment model" components, in particular the link between the perceptions and the symbolic state contained into the environment model. We share the idea of anchoring low-level sensory perceptions and high-level symbolic representation proposed in [21], and further studied in [77, 97, 44, 88, 29, 89]. In [21], authors provide a general overview of the anchoring problem, and discuss the main challenges that arise when building a robotic system that requires perceptual anchoring. For example, at the symbolic level, we can identify objects with a specific name (e.g.

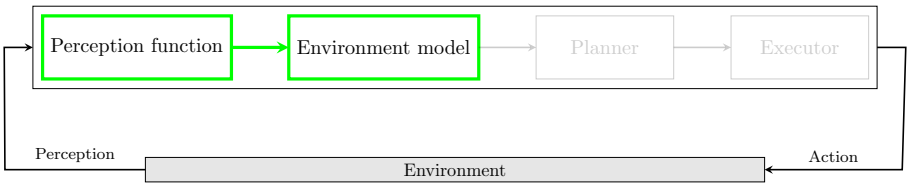


Figure 3.1: Perceptual anchoring in the agent environment interface.

"$block_0$"), while the perceptual system is not in general able to perceive the identity of an object but only some of its properties (e.g. the color of the block, its size, etc.). In [77], the anchoring framework introduced in [21] is extended for tackling the problem of maintaining coherent perceptual information in a mobile robotic system working over extended periods of time, i.e., to cope with perception management considering multi-sensing resources and temporal factors. The perceptual anchoring framework proposed in [21] uses a top-down approach, from symbols to perceptions, while in [77] the approach is extended to be also bottom-up, i.e., from perceptions to new symbols. Moreover, [77] focus on the problem of maintaining anchors, that is, matching them when an object has already been seen in the past, and deleting them when their "life" time is expired and the object probably removed from the environment. [97] exploits perceptual anchoring for creating a graph-based model with object unary features (e.g. the position of an object) and pairwise features (e.g. the perpendicularity between two objects). The graph-based model is then combined with a probabilistic graphical model [63] for extracting contextual relations between objects that are used as additional information for object recognition. A follow-up work is done in [44], where the approach learns the function for matching a new anchor with a previously seen one by means of a Support Vector Machine (SVM). The SVM classifier is trained on samples of object pairs manually labeled as "same or different object", in order to approximate the similarity between two objects. In [88, 89], the bottom-up approach proposed in [77] is deepened for dealing with semantic relational object tracking. Interestingly, as a perception function, authors exploit a predicate grounding relation given as input for predicting properties of objects (e.g. the color of a cup), and spatial relationships between objects (e.g. an apple is behind a cup). Furthermore, they enhance the anchoring process with high-level probabilistic reasoning for tracking and predicting the state of objects that might not be perceived due to, e.g., occlusion. In [29], the work proposed in [88, 89] is extended for dealing with multi-modal probability distributions and integrated with statistical relational learning to learn probabilistic logic rules for reasoning about the objects state. All the aforementioned approaches tackle the anchoring problem per se, whereas in this work, we consider a broader framework integrating perceptual anchoring with symbolic planning and execution, for enabling agents to either perform tasks in unknown environments or plan for learning object anchors, i.e. object properties.

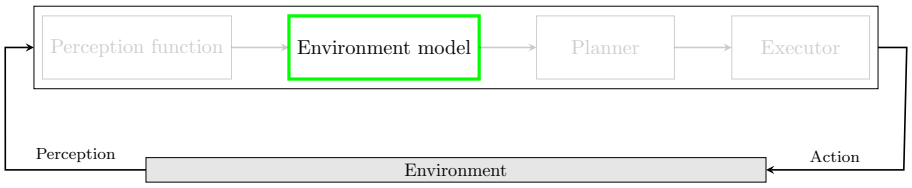


Figure 3.2: Action model learning approaches in the agent environment interface.

3.2 Action model learning

In the planning literature, there are many works that aim to learn a symbolic representation of the planning domain from sequence of symbolic states corresponding to transitions. The approaches can be grouped into offline and online. In the offline setting, a set of plan traces, where each plan trace consists of subsequent transitions, is given as input; and the preconditions and/or effects and/or input parameters of the actions are inferred from the set of plan traces. In the online setting, there is no set of input plan traces, and the agent has to either generate informative plan traces online or execute the actions selected according to some criteria (e.g. randomly or by a human teacher).

3.2.1 Offline approaches

Recent offline approaches address the problem of model learning with different assumptions on the observability of states and actions.

Fully observable plan traces

[109] proposes an approach for learning grounded action models from a set of fully observable plan traces. They learn action preconditions and effects by looking at the transitions in each plan trace. Interestingly, the learned action model is guaranteed to be safe and produce sound plans. The approach above has been extended in [56] for learning lifted action models, where authors propose a method called Safe learning of lifted Action Models (SAM). The learned action models preserve the theoretical property of safety, but an additional assumption called injective action binding assumption is introduced, i.e., the assumption that every action parameter is mapped to a different object. Notably, in [56], an Extended version of SAM (E-SAM) is proposed for dealing with ground actions that do not respect the injective action binding assumption. A follow-up work [55] extends SAM for learning probabilistic action models which are probably safe and approximately complete. With respect to all the aforementioned approaches, SAM and its variants require fully observable plan traces. In [12], authors propose a method for learning action models from the structure of the state space associated with small problem instances. Notably, they do not assume knowledge of the action schemas, predicate symbols, or objects, which

are learned from the input state space. In particular, they learn action models that produce state-space graphs isomorphic to the input ones, by encoding the learning problem as a SAT problem. However, in [12], the input graphs are assumed to be complete and without noise. A follow-up work [94] relaxes these assumptions, where authors propose a more efficient encoding of the learning problem in answer set programming.

Partially observable plan traces

Action Relation Modeling System (ARMS) [124] is one of the first approaches for learning STRIPS action models from plan traces with partially observable states. For learning the action model, ARMS builds a weighted propositional satisfiability (weighted MAX-SAT) problem and solves it using a MAX-SAT solver. The constraints of the SAT problem are extracted from the states and actions in the plan trace (e.g. by looking for frequent relation-action pairs). Finally, ARMS guarantees that the learned action models are approximately correct and concise. The notion of correctness and conciseness are given according to evaluation metrics (i.e. error and redundancy) defined in [124]. Learning Object Centred Models (LOCM) [24] learns classical planning action models from plan traces with partially observable states. Particularly, LOCM groups the objects according to their position in the ground action names contained in the plan traces. Afterward, it assembles the transition behavior of each group of objects, the co-ordinations between transitions of different groups of objects, and the relationships between objects of different groups. To achieve this, LOCM relies on the assumption that actions change the state of objects, and whenever an action is executed, the preconditions and effects on an object are the same. The learned models are specified in the form of parameterized finite state machines, where the state parameters encode associations between objects. Notably, LOCM does not require input knowledge about the planning domain to be learned (e.g. the set of predicate names), however, it does not deal with static knowledge (e.g. static preconditions), which needs to be explicitly specified. A follow-up work (LOCM2) has been done in [23], where LOCM has been extended for learning a wider range of domains by allowing a group of objects to be represented by multiple parameterized finite state machines. Finally, LOP [43] extends LOCM for dealing with static knowledge, but requires additional input knowledge (i.e. a set of optimal plans). [129] introduces an approach, namely Learning Action Models from Plan traces (LAMP), that learns action models with quantifiers and logical implications from a set of plan traces with partially observable states. They first encode the state transitions in the plan traces into propositional formulas, then they generate candidate formulas involving the input predicate list and domain constraints. Next, they build a Markov Logic Network [91] and select the logical formula with the higher weight. Finally, the selected formulas are converted into the learned action model. However, the learned action models are neither correct nor complete. Another prominent system is Fama [4], which learns action models offline from examples by transforming the learning task into a classical planning task. It

works with different kinds of inputs, from a set of plans to just a pair of initial and final states, without intermediate actions or states. Moreover, it accepts in input partially specified action models. On the one hand, the aforementioned approaches to offline learning can deal with partial observability of states and actions, and some of them even with noisy states and noisy actions.

Noisy plan traces

[128] proposes a method for learning action models from plan traces with partially observable states and noisy actions, namely Action Model Acquisition from Noisy plan traces (AMAN). In particular, AMAN looks for the action model that best explains the input plan traces. To find such a model, firstly a set of possible action models is computed, then AMAN creates a graphical model with the causal relations between states, actions (grouped into correct actions and probably noisy actions), and possible action models. Finally, AMAN exploits the plan traces and returns the model that maximizes a reward function defined in terms of the percentage of actions successfully executed and the percentage of goal propositions achieved after the last successfully executed action. In [85], authors propose a method for learning action models from a set of plan traces with partially observable and noisy states, where the execution of actions in the traces can also fail. They encode the symbolic state as binary vectors, then compute the difference vector between the starting and destination states in the transitions contained in the plan traces. Next, for each action and propositional atom, they train a voted perceptron classifier [37] that predicts the difference value of the propositional atom in the difference vector, given a starting state and the action associated with the classifier. Afterward, a set of logical rules representing preconditions and effects is extracted from the classifier associated with each action. Finally, rules are filtered and combined to produce a STRIPS action model. All these approaches are offline, require input plan traces that in some cases might be not available, and hence do not deal with the issue of selecting informative plan traces.

3.2.2 Online approaches

Since the seminal work on online learning of operators [40, 41, 116], and the first approaches for learning action models by integrating learning, planning, and execution [38], some recent approaches have addressed the problem of online and incremental learning of action models. [114] proposes an approach to online learn action models which can be used in web-service planning problems. Their approach requires the use of an external "teacher" providing plan traces on demand. 3SG [15] is an online algorithm that learns probabilistic action models with conditional effects and deals with action failures, sensory noise, and incomplete information. [120] describes an instance-based online method for learning action models in relational domains. The work is extended to deal with both discrete and continuous action models [121, 122]. [92] propose a technique based on relational RL to learn deterministic action models, and [93]

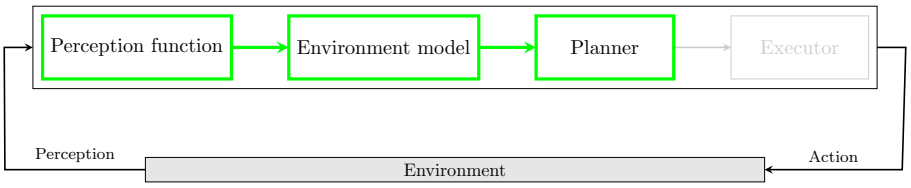


Figure 3.3: Planning in a latent space within the agent environment interface.

extend the approach to deal with nondeterministic actions. These approaches are based on important technical differences with respect to our work. Most importantly, the main conceptual and practical difference is that all these approaches assume that the action to be executed is randomly selected or given as input, and therefore do not deal with the problem of guiding the exploration phase toward informative states.

3.3 Planning in a latent space

There are some approaches that plan starting from continuous observations (e.g. RGB images), rather than symbolic states. They map each continuous observation into a symbolic state composed by a set of latent boolean variables, which is suitable for symbolic reasoning. These approaches focus on learning the perception function, environment model components, and exploit the learned representations for applying symbolic planning, as shown in Figure 3.3.

LatPlan [7, 6, 8] takes as input pairs of high dimensional raw data (e.g., images) corresponding to transitions and learns both a symbolic planning domain and a mapping among images and symbolic states. In particular, the mapping among images and symbolic states is learned through a variational autoencoder [59], which takes as input an image of the environment and finds the corresponding propositional state representation in a latent space; a similar approach has been proposed in [3]. The learned autoencoder is used to convert pairs of subsequent images to symbolic transitions, from which an off-the-shelf action model learning method generates a ground action model. Specifically, in the learned action model, the ground action parameters, preconditions, and effects are specified in terms of latent variables. For solving a planning problem, LatPlan takes as input an initial and goal state image, converts them into symbolic states with the learned autoencoder, and computes a plan with the learned action model. Finally, intermediate states in the plan are decoded back to a human-comprehensible image sequence. LatPlan is an offline approach, while our approaches [69, 71] are online and work also in dynamic environments.

In [46], authors propose a model-based method for learning a latent dynamics model from image observations and planning in the latent space. With respect to LatPlan, they do not exploit the learned latent representation for applying symbolic planning, since planning is performed by means of model-predictive control [90].

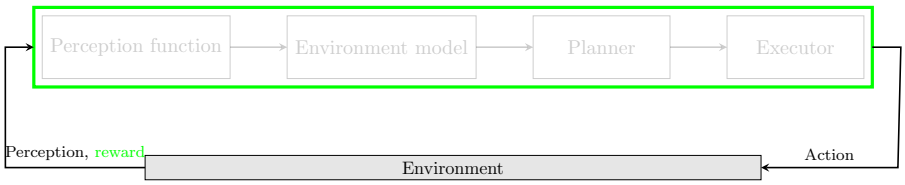


Figure 3.4: Model-free Reinforcement Learning agent environment interface.

Causal InfoGAN [68] learns discrete or continuous models from high dimensional sequential observations. This approach fixes a priori the size of the discrete domain model. Differently from our approaches [69, 71], their goal is to generate an execution trace in the high dimensional space. In particular, given a pair of initial and goal observations, they compute a sequence of observations from the initial observation to the goal one, such that the computed sequence can be observed by executing actions. This is achieved by projecting an initial observation and a goal observation into the corresponding states in a discrete planning domain, computing a plan, and exploiting a Generative Adversarial Network (GAN) for mapping the states encountered by executing the plan into image observations.

On the one hand, the above approaches deal with the problem of mapping continuous observations into symbolic states. Moreover, for planning domain states that can be visually represented, these approaches can produce a visual plan that is human-comprehensible. On the other hand, the state variables are latent, and the action preconditions and effects specified in the learned action models are not human-readable, which is a critical limitation when there is a need of understanding how the environment is affected by action execution.

3.4 Planning by Deep Reinforcement Learning

The approaches based on RL [57, 110] focus on learning policies, and typically assume the set of states and the correspondence between continuous data from sensors and states is fixed and given. There are, however, RL approaches that plan and learn directly in a continuous space, i.e. approaches based on deep RL. These approaches exploit deep learning models for learning an embedding of the agent state, represented by a sequence of continuous features, starting from continuous observations. RL approaches can be mainly grouped into model-free and model-based. In model-free approaches, the agent has no internal model of the environment, and directly learns from experience, i.e. by executing actions. Whereas in model-based RL, a model of the environment's dynamics is learned and used to supplement direct learning from experience.

3.4.1 Model-free

In model-free deep RL, none of the components shown in Figure 1.2 is given as input, i.e., all of them are learned simultaneously. Figure 1.2 is similar to the agent-environment interface of an RL agent, except for the information perceived by the agent, which does not only consist of sensory data but also of a reward. In particular, in model-free deep RL, all the agent components in Figure 1.2 are learned by end-to-end training, as shown in Figure 3.4. On the one hand, deep RL techniques make it easier to develop agents that solve difficult tasks, without the need of learning each component, this can be particularly suitable for solving low-level control tasks such as grasping objects with a robotic hand. On the other hand, the harder the task to be solved the more the data required for training an RL agent, which can be a limitation in scenarios where, e.g., few data are available and/or there are no simulators. Moreover, there are some well-known drawbacks in deep RL: *(i)* lack of explainability about why a deep RL agent decides to execute a particular action; *(ii)* the reward function depends on a specific goal, i.e., once an RL agent is trained for solving a particular task (e.g. moving blocks on a table), it is difficult to generalize over different tasks (e.g. moving balls among different rooms) without changing the reward function and retraining the model. A breakthrough in deep RL has been the work by [84], where authors introduced deep-Q learning, a deep RL algorithm that can learn successful policies directly from high-dimensional sensory inputs using end-to-end RL. A follow-up work is proposed in [113], where some drawbacks about the value overestimation of deep Q-learning in large-scale problems are overcome by integrating deep Q-learning and double Q-learning [48]. Another deep RL breakthrough is presented in [75], where the deterministic policy gradient method [105] is integrated with deep Q-learning for dealing with continuous action spaces. Such approaches are very suited to address some tasks, e.g., moving a robot arm to a desired position or performing some manipulations. However, we believe that, in several situations, it is conceptually appropriate and practically efficient to learn an abstract discrete and deterministic model where planning is much easier and more efficient to perform. Notably, the aforementioned approaches perform at a human level in a wide range of games and continuous control tasks. However, they assume the environment is fully observable, and do not tackle the problem of abstracting sensory data (i.e. images) into a symbolic and explainable representation.

Object Goal Navigation

In the context of Embodied AI [30], the proposed approaches are mostly based on deep RL. As an example, we consider the object goal navigation task, where a robotic agent is placed in a random position of an unknown environment (e.g. an apartment) and asked to find and go close to an object instance of a given goal object type. In particular, this task falls under the umbrella of visual navigation and exploration. To the best of our knowledge, the majority of the proposed approaches are based on deep RL. [83] formulates the problem

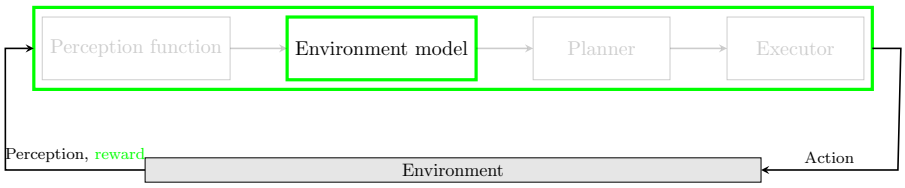


Figure 3.5: Model-based Reinforcement Learning agent environment interface.

of navigating in complex environments with dynamic objects as an RL problem. In particular, they jointly learn the goal-driven RL policy with auxiliary depth prediction and loop closure classification tasks. The auxiliary tasks are used to enrich the (latent) representation learned during training and improve data efficiency. [17] proposes a deep RL method, namely Active Neural SLAM (ANS), for learning a policy for navigating in an unknown environment and optimizing the exploration. Precisely, they construct a topological map of the environment from depth observations; then an RL algorithm is applied on such a map, with the objective of learning a policy that selects a point, reached via path planning, to maximize the environment exploration. Afterward, same authors [18] extend the approach to cope with semantic exploration. The occupancy map is enriched with semantic information about objects in the scene. The policy, trained specifically for solving the object goal navigation problem, exploits the semantic information available in the map. [34] learns a memory-based policy, which constructs semantic maps by means of an encoder-decoder model with a spatial memory transformer. The memory-based policy embeds and adds each observation to a memory and uses the attention mechanism to exploit spatiotemporal dependencies. They experimentally evaluate the learned policy by solving the object goal navigation problem. In [86], they propose a similar approach that learns a navigation policy through deep RL by using as a visual representation of the state the semantic segmentations and detection masks provided by off-the-shelf segmenter and detector. Another similar approach is proposed in [126], where authors enrich the learned representation by means of auxiliary tasks, e.g., the task of predicting the action executed by the agent given two subsequent observations.

3.4.2 Model-based

An alternative approach in RL is model-based deep RL. With respect to Figure 3.4, in model-based deep RL the agent component environment model is learned (Figure 3.5). In particular, the environment model is a Markov Decision Process (MDP) used for estimating the state values and then learning the policy accordingly. On the one hand, model-based algorithms are known in general to outperform model-free ones in terms of sample complexity [26], since they do not need to generate trajectories by executing actions online, but they can generate trajectories in the learned MDP model. On the other hand, typically model-based deep RL requires more computational resources w.r.t. model-free

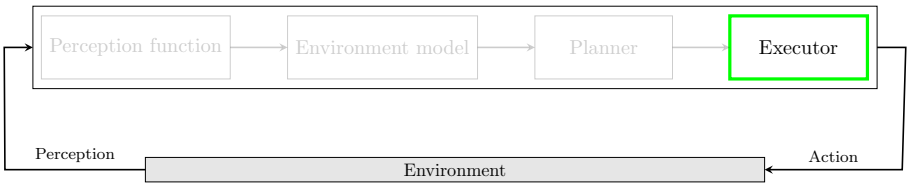


Figure 3.6: Agent environment interface with a focus on the integration of symbolic planning and deep reinforcement learning.

deep RL, which makes model-based deep RL unsuitable when dealing with high-dimensional state representations. One of the most recent breakthroughs in deep RL has been proposed in [106], where authors trained a deep RL agent for playing Go and won against the human European Go champion. In particular, in [106], both a policy network and a value network are trained, and then combined with Monte Carlo Tree Search [22, 62]. The approach has been extended in [102] for playing other games obtaining superhuman performance equivalent to the model-free approach proposed in [106]. A discussion about peculiarities and drawbacks of [102] has been reported in [47]. We share the idea of a planning domain at the abstract level with all the work on abstraction on MDP models, see, e.g., [1, 74, 65, 2]. However, our problem and approach are substantially different, since in the work about abstraction on MDP models, the mapping between original MDP states and abstract states is given, while we learn it.

Our approach shares some similarities with the work on planning by RL, since we learn by acting in the environment. However, in RL, the agent directly perceives the state of the environment, and there is no need to learn a perception function that links the sensory data with the abstract states of the environment. Even though in deep RL a perception function is learned, the state representations are not human-comprehensible, and it is not trivial to explain why an agent decides to execute an action. Furthermore, the learned perception function provides state representations that strictly depend on the goal, i.e., the perception function learned when solving a specific task is not likely to generalize to different tasks.

3.5 Symbolic Planning and Deep Reinforcement Learning

The problem of integrating symbolic action models with low level sensory data and actions has been addressed by different approaches, i.e., the problem of learning the executor component in Figure 3.6. Most of the proposed approaches are based on RL techniques. [78] proposes a framework, called Symbolic Deep RL (SDRL), which combines symbolic planning and deep RL to improve the data efficiency and interpretability of deep RL, and learn policies that compile high-level actions into low-level operations. Their goal is to learn both a

sequence of subtasks, which are symbolic actions, and the corresponding sub-policies, so that executing the sub-policy for each subtask one by one can achieve the maximal cumulative reward. To this aim, they propose a formulation that maps symbolic transition to a similar structure of RL options [66]. SDRL assumes that a grounded domain model is provided in input and never updated, i.e., it does not deal with the problem of learning the environment model component in Figure 1.2. Moreover, SDRL assumes a perfect oracle that maps low-level perceptions into symbolic states, i.e., it assumes a perfect perception function component (Figure 1.2) is given as input.

Neural Symbolic RL (NSRL) [79] represents abstract domains in first-order logic and uses RL to learn high-level policies. NSRL generates a compact representation of the learned policies as a set of rules via inductive logic programming. In particular, NSRL encodes the symbolic states into state matrices, then applies an attention mechanism to weight the more relevant facts in each state, i.e., the more relevant rows in each matrix. Then, NSRL multiplies the weighted state matrices to generate logical rules, according to a process called multi-hop reasoning, which is described in [125]. Afterward, another attention mechanism is applied to the logical rules to associate them with weights, and the weighted logical rules are given as input to a multilayer perceptron network that outputs the action to execute. Similarly to SDRL, NSRL assumes a given and fixed abstract domain instantiation and a perfect mapping from sensory data to symbolic states.

DPDL [58] represents abstract domains in PDDL. It learns online both mappings from sensory data to symbolic states (i.e. the perception function) and low-level policies for executing high-level actions. This allows DPDL to generalize over different tasks by reusing the learned low-level policies. DPDL learns online the mappings from sensory data to symbolic states (i.e. the perception function), and separately learns a low-level policy that translates symbolic operators into executable actions on the robot (i.e. the executor). Both the perception function and low-level policy are modeled as variational temporal convolutional network [73] that takes as input a finite sequence of observations (i.e RGB images). The perception function returns the truth values of the propositional atoms describing the symbolic state. The low-level policy takes as input also the symbolic operator to be executed and outputs a target configuration in the joint space. As the other methods mentioned above, DPDL assumes a given and fixed grounded PDDL domain.

Chapter 4

Learning Planning Domains from Sensor Data

The specification of planning domains is a challenging task. A good planning domain should abstract away the details of the world state which are irrelevant to the achievement of the agents' goals, keeping only the relevant details. In many real applications, it may happen that agents do not have a good planning domain in advance. For instance, a robot moving packages among the rooms of a building could have a model of the map of the building with a number of flaws or relevant missing details. In these cases, agents should be able to learn and update their models (planning domains) while acting in the world and observing the consequences of their actions. This is the main purpose of the PAL algorithm (Planning, Acting, and Learning) proposed in [103], which learns, incrementally and online, a discrete deterministic planning domain from real-value observations of the world. Each domain state is linked to observations by the so-called *perception function*, which provides the likelihood of the observations when the agent is at that specific state. At each iteration, PAL updates the set of states of the extensional representation of a planning domain, possibly by introducing new states for unexpected observations, and it adjusts the transition relation and the perception function.

In order to overcome the scalability limitations of PAL, the agent is provided with an initial "draft" planning domain specified in PDDL [81]. This domain is required to be neither complete nor correct. This planning domain is used to *guide the agent in the discovery* of the world. We propose a new algorithm, that learns the extensional planning domain and incrementally updates and corrects the initial PDDL domain with additional information collected during the execution of actions.

In the proposed approach, the presence of both the extensional and the PDDL planning domain is exploited to efficiently achieve the agent's goals through two alternative and complementary planning algorithms: (i) a shortest-path algorithm for planning in the space of the states of the extensional model

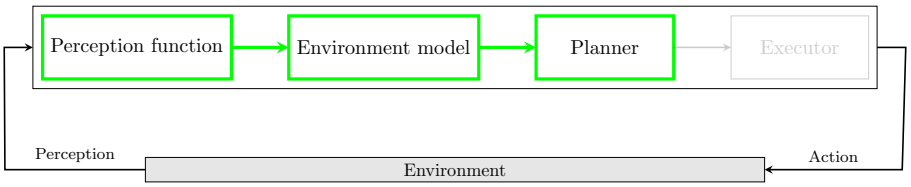


Figure 4.1: PAL within the agent environment interface.

(i.e. the states already discovered by the agent), and (ii) a PDDL planning algorithm for generating plans that allow discovering new states, if the goal is not achievable in the extensional model.

Considering the agent-environment interface introduced in Chapter 1, in this Chapter we focus on: *(i)* learning the perception function, since the agent maps the low-level perceptions into the high-level states of the planning domain; *(ii)* learning the environment model, which is the extensional model learned by the agent; and *(iii)* the planner module, given that we plan with both the extensional and intensional models, as shown in Figure 4.1.

4.1 The Plan-Act-Learn Problem

A first brief and intuitive overview of the PAL problem is shown in Figure 4.2. At the beginning, the agent perceives the environment and associates the perception with a new state of the planning domain. a set of sensors (e.g. odometric sensors and RFIDs). Firstly, the agent perceives the environment and builds a symbolic description of the environment about its current state. Next, it plans to achieve a given goal starting from the symbolic representation of its current state built so far. Then, the agent executes the plan, and after each action execution, it updates the perception model used to map sensory data into abstract states, and the extensional model of the planning domain, which can be used for further planning.

A PAL-problem instance consists in learning an abstract model of the environment that can be exploited by an agent to achieve a set of goals. In the PAL problem, agents perceive the environment through a series of *perceptions*, where a perception is a vector $\mathbf{x} = \langle x_1, x_2, ..., x_n \rangle$ of continuous real value variables, called *perception variables*. We define the environment where agents operate as a non-deterministic infinite-state transition system, called *perceptible environment*.

Definition 7 (Perceptible environment)**.** *A perceptible environment $\mathcal{E}$ is a tuple (Q, A, τ), where $Q \subseteq \mathbb{R}^n$ is a (possibly infinite) set of perceptions, A is a finite set of actions, and $\tau : Q \times A \to 2^Q$ is a non-deterministic transition function.*

Function τ returns the set of possible perceptions after the execution of an action $a \in A$ in a state $q \in Q$ (and before executing other successive actions). We adopt the notation $\tau(a, X) = \bigcup_{\mathbf{x} \in X} \tau(a, \mathbf{x})$ for $X \subseteq Q$.

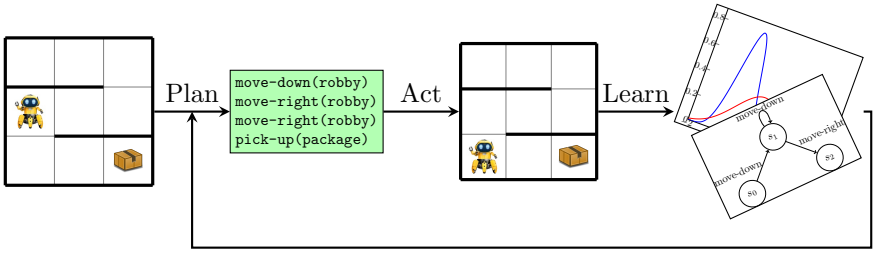


Figure 4.2: PAL problem overview

Specifying the components of a perceptible environment is typically extremely complicated, and it cannot be done by hand. In the field of planning, a common assumption is that agents act at an *abstract* level. For instance, the behavior of a robot moving packages among the rooms of a building can be conveniently determined by a planning domain where each state corresponds to the fact that the robot and packages are in a certain room, and each transition corresponds to an abstract action, such as moving the robot among rooms, picking up packages, and putting down them. We define the search space for planning as a deterministic finite-state transition system.

Definition 8 (Extensional model)**.** *An* extensional model $\mathcal{M}$ *of an environment* $\mathcal{E} = (Q, A, \tau)$ *is a tuple* (S, A, γ) *where* S *is a finite set of (abstract) states, and* $\gamma : S \times A \to S$ *is a deterministic transition function.*

Given a state $s \in S$ and an action $a \in A$, the function γ outputs the resulting state reached after the execution of a in s. The action space A of the extensional model is the same as of the perceptible environment, which consists of the set of actions agents can perform.

Definition 9 (Perception function)**.** *Given an extensional model* $\mathcal{M} = (S, A, \gamma)$ *of an environment* (Q, A, τ)*, a perception function* ρ *for* $\mathcal{M}$ *is a function* $\rho : Q \times S \to \mathbb{R}^+$ *such that for every* $s \in S$*,* $\rho(\mathbf{x}, s) = p(\mathbf{x} \mid s)$*, where* $p(\mathbf{x} \mid s)$ *is a probability density function on* Q*.*

The extensional model $\mathcal{M}$ and the perception function ρ share the same set of states S. Given a perception function ρ and a perception $\mathbf{x} \in Q$, we define the function $\rho^* : Q \to S$ as $\rho^*(\mathbf{x}) = \text{argmax}_{s \in S}\, \rho(\mathbf{x}, s)$, and similarly $\rho^*(X) = \{\rho^*(\mathbf{x}) \mid \mathbf{x} \in X\}$. Intuitively, ρ^* is the function that discretizes the infinite set of states Q into the finite set of states S.

Definition 10 (Plan)**.** *A* plan *in an extensional model* $\mathcal{M} = (S, A, \gamma)$ *from state* $s \in S$ *to state* $s' \in S$ *is a sequence* $(a_1, \ldots, a_m)$ *of* m *actions in* A *such that* $s' = \gamma(a_m, \gamma(a_{m-1}, \ldots, \gamma(a_1, s)))$*.*

A perception goal is a perception $\mathbf{x} \in Q$ that, when perceived by the agent, makes it consider the assigned task accomplished.

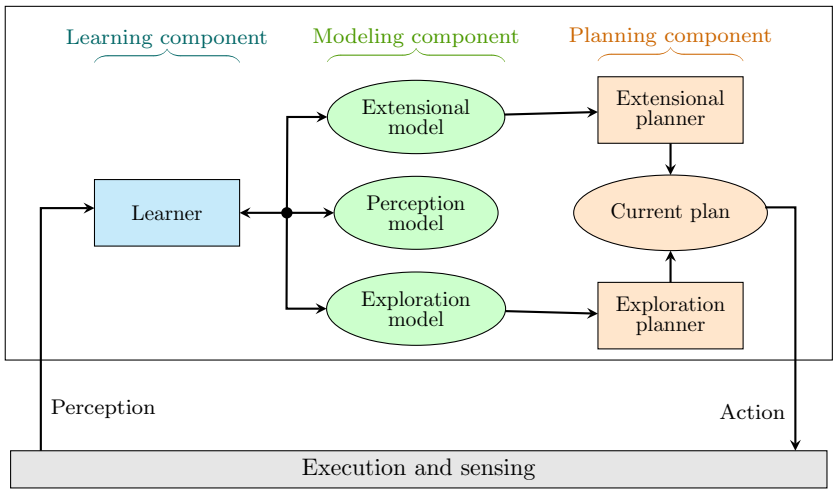


Figure 4.3: PAL architecture. The square boxes represent modules; the circle ones represent data.

Definition 11 (PAL-problem)**.** *Given an environment $\mathcal{E}$, the* PAL-problem *consists in learning an extensional model $\mathcal{M}$ and a perception function ρ from $\mathcal{E}$, such that, for every perception $\mathbf{x}_0 \in Q$ and (non-empty) perception goal set $X_g \subseteq Q$, $\mathcal{M}$ has a plan $(a_1, \ldots, a_m)$ from $\rho^\star(\mathbf{x}_0)$ to some state in $\rho^\star(X_g)$ and $\tau(a_m, \tau(a_{m-1}, \ldots, \tau(a_1, \mathbf{x}_0))) \cap X_g \neq \emptyset$.*

Note that the agent does not know the environment $\mathcal{E}$. The only knowledge about the environment that it has is the one observed through the perception variables when executing actions, as the agent can only perceive the environment and observe the action effects after their execution.

4.2 Solving the PAL Problem

We introduce an approach for solving the PAL problem that interleaves planning, acting, and learning using a limited amount of prior knowledge for the agent. Our approach is named as the problem it solves, Plan-Act-Learn (PAL). To learn the extensional model, the agent can apply different strategies: a random exploration strategy is not feasible, since, as shown in [103], it does not scale to large state spaces. Alternatively, the agent can use some prior *belief* about the environment to decide a plan that will lead to its current goal. Following this idea, we suppose that such a belief is expressed through an *exploration planning domain* $\mathcal{D}^e$ that is specified by a planning language such as PDDL [81]. Intuitively, the agent will decide the next action to perform by computing a plan that reaches a state among those in an input set of goal states G^e from the PDDL state s^e representing the belief of the agent about the current status of the environment (the e index indicates that these are the initial state and set

of goal states of the exploration model).

Note that we make no assumption about the correctness of $\mathcal{D}^e$; we only need that the transformation used to derive s^e from the current status of the environment is such that the inverse transformation applied to a goal state among those in G^e derives a status of the environment corresponding to a perception in X_g.

The architecture of the proposed solution is shown in Figure 4.3. The top box of the picture shows the architecture of the PAL agent. It consists of three components: (i) the learner module, (ii) the modeling component, formed by three models, and (iii) the planning component using two kinds of planners. The *learner* updates the perception model using the perceptions from the environment, incrementally constructs the extensional model, and revises the exploration model. We call the perception function, together with the history of sensed perceptions, the *perception model*. The exploration model is refined when a failure occurs in the attempt of executing an action. The extensional model and the exploration one are respectively taken as input by the extensional and exploration planner. Firstly, a plan is searched in the extensional model. The search for such a plan may fail because the transition function γ known by the agent at planning time could be incomplete. If no plan is found, the goal-driven exploration strategy is applied by means of an exploration planner. The PAL agent interacts with a simulator (bottom box in the picture), the purpose of which is to simulate the perceptible environment $\mathcal{E}$ where the agent operates, i.e., it simulates the execution of a given action and the sensing of the environment immediately after the action execution. We assume that the simulator knows the comprehensive definition of the transition function τ of $\mathcal{E}$.

The pseudocode of PAL is shown in Algorithm 1. The algorithm takes as input: a perception goal set X_g, a threshold $t \in \mathbb{R}$, an initial perception function ρ_{init}, an initial extensional model $\mathcal{M}_{init}$, and an initial exploration model. The input initial extensional model $\mathcal{M}_{init}$ is composed of a (possibly empty) set of states S_{init}, a (possibly empty) transition function γ_{init}, and a set of actions A that agents can perform; the input initial exploration model is composed by $\mathcal{D}^e_{init}$, s^e, and G^e. Initially, the agent perceives the environment by sensing perception $\mathbf{x}$ (Line 2). Afterward, it verifies whether there exists at least a state among those in S_{init} such that the likelihood of sensing $\mathbf{x}$ being in this state is greater than a threshold. If it does exist the current state s is set to $\rho^*(\mathbf{x})$, otherwise a new state is created (Lines 3–8). The perception history is initialized with the perception $\mathbf{x}$ and the current state s (Line 10). If the perception $\mathbf{x}$ belongs to the set X_g, then s is a goal state and the algorithm returns success (Lines 12–13). Otherwise, if the plan π is empty, the set of goal states S_g is defined as the states in S which correspond to a goal perception $\mathbf{x}_g \in X_g$ (Lines 15–16). A state s corresponds to a goal perception $\mathbf{x}_g$ if $\rho(\mathbf{x}_g, s) \geq t$. If the set S_g is not empty, the extensional planner is exploited to search a plan π from s to a state in S_g (Lines 17–18). The extensional planner runs the Dijkstra algorithm on the graph induced by the state transition function γ, to find the shortest-path from the current state s to a goal state in S_g defined in Line 16.

Algorithm 1 PAL Algorithm

Input: $X_g, t, \rho_{init}, \mathcal{M}_{init}, \mathcal{D}^e_{init}, s^e, G^e$.

1: $\rho, S, \gamma, \mathcal{D}^e, \pi \leftarrow \rho_{init}, S_{init}, \gamma_{init}, \mathcal{D}^e_{init}, \langle\rangle$
2: $\mathbf{x} \leftarrow$ SENSE()
3: $S' \leftarrow \{s \in S \mid \rho(\mathbf{x}, s) \geq t\}$
4: **if** $S' = \emptyset$ **then**
5: $\quad s \leftarrow$ CREATESTATE($\mathbf{x}$)
6: $\quad S \leftarrow S \cup \{s\}$
7: **else**
8: $\quad s \leftarrow \rho^\star(\mathbf{x})$
9: **end if**
10: $P \leftarrow \langle \mathbf{x}, s \rangle$
11: **while** the CPU time limit is not exceeded **do**
12: $\quad$ **if** $\mathbf{x} \in X_g$ **then**
13: $\quad\quad$ **return** *Success*;
14: $\quad$ **end if**
15: $\quad$ **if** $\pi = \langle\rangle$ **then**
16: $\quad\quad S_g \leftarrow \{s \in S \mid \rho(\mathbf{x}_g, s) \geq t$ and $\mathbf{x}_g \in X_g\}$
17: $\quad\quad$ **if** $S_g \neq \emptyset$ **then**
18: $\quad\quad\quad \pi \leftarrow$ EXTENSIONALPLANNER(γ, s, S_g)
19: $\quad\quad$ **end if**
20: $\quad\quad$ **if** $\pi = \langle\rangle$ **then**
21: $\quad\quad\quad \pi \leftarrow$EXPLORATIONPLANNER($\mathcal{D}^e, s^e, G^e$)
22: $\quad\quad$ **end if**
23: $\quad$ **end if**
24: $\quad$ **if** $\pi \neq \langle\rangle$ **then**
25: $\quad\quad \pi \leftarrow$POP($\pi$)
26: $\quad$ **else**
27: $\quad\quad$ Select an action $a \in A$ randomly
28: $\quad$ **end if**
29: $\quad$ EXECUTE(a)
30: $\quad \mathbf{x} \leftarrow$SENSE()
31: $\quad S' \leftarrow \{s \in S \mid \rho(\mathbf{x}, s) \geq t\}$
32: $\quad$ **if** $S' = \emptyset$ **then**
33: $\quad\quad s' \leftarrow$ CREATESTATE($\mathbf{x}$)
34: $\quad\quad S \leftarrow S \cup \{s'\}$
35: $\quad\quad \gamma \leftarrow \gamma \cup \{(s, a, s')\}$
36: $\quad\quad s^e \leftarrow$UPDATEPDDLSTATE($\mathcal{D}^e, s^e, a$)
37: $\quad$ **else**
38: $\quad\quad s' \leftarrow \rho^\star(\mathbf{x})$
39: $\quad\quad$ **if** $s' = s$ **then**
40: $\quad\quad\quad \mathcal{D} \leftarrow$UPDATEPDDLDOMAIN($\mathcal{D}^e, s^e, a$)
41: $\quad\quad\quad \pi \leftarrow \langle\rangle$
42: $\quad\quad$ **else**
43: $\quad\quad\quad s^e \leftarrow$UPDATEPDDLSTATE($\mathcal{D}^e, s^e, a$)
44: $\quad\quad$ **end if**
45: $\quad$ **end if**
46: $\quad s \leftarrow s'$
47: $\quad P \leftarrow$APPEND($P, \langle \mathbf{x}, s \rangle$)
48: $\quad \rho \leftarrow$UPDATEPERCEPTIONFUNC(ρ, P)
49: **end while**
50: **return** *Failure*

If the extensional planner does not find a plan, the agent searches for a plan using the exploration model, i.e., it runs a PDDL planner to achieve goals G^e from s^e, using the PDDL domain $\mathcal{D}^e$ (Lines 20–21). Note that, since the

exploration model is an approximation of the agent behavior in the real world, it could happen that also this plan does not exist or, if it does exist, an action in the plan is not executable in the real world. If the plan does not exist, an action in A is randomly selected (Line 27). Otherwise, the first action of the plan is selected (Lines 24–25) and executed (Line 29). After the execution of the action (and before executing the next one), the agent perceives the current environment state (Line 30). Notice that different sensing can correspond to the same abstract state. This can happen for several reasons. For instance, two very closed GPS perceptions (of a robot being in the same room) correspond to the same abstract state. Furthermore, noisy perceptions done by the agent while being idle should be mapped in a unique abstract state.

Given the last executed action a and the last perception $\mathbf{x}$, the learner updates the perception model, the extensional model, and the exploration model. Specifically, if the probability of observing $\mathbf{x}$ from each state in S is lower than threshold t, then the agent creates a new state s', adds s' to S, adds transition $\langle s, a, s' \rangle$ to γ and updates the PDDL state s^e (Lines 32–36). The state s^e is updated according to $\mathcal{D}^e$, i.e., adding the positive effects and deleting the negative effects of action a. Otherwise, the agent selects the state s' that maximizes the likelihood of observing $\mathbf{x}$ as the next state (Line 38). If the states in S that maximize the likelihood of observing $\mathbf{x}$ are more than one, one of them is randomly selected. If $s = s'$, i.e., the execution of action a fails, then the agent makes plan π empty and updates the PDDL domain $\mathcal{D}^e$ in such a way that action a cannot be executed in state s^e (Lines 39–41). If the action has been successfully executed, the PDDL state is updated by applying the action effects (Line 43). Finally, the current state s is set to the next state s', the pair $(\mathbf{x}, s)$ is added to the perception history P, and the perception function ρ is updated according to P (Lines 46–48). The loop 11–49 is repeated until the CPU time limit is exceeded. If the loop terminates without having reached a goal state, the algorithm returns failure (Line 50).

4.3 Learning the Perception Function

The perception function ρ allows the agent to map a value $\mathbf{x} = (x_1, \ldots, x_n)$ of n perception variables to the state s^* according to the maximum likelihood criteria $s^* = \text{argmax}_{s_i \in S} \rho(s_i, \mathbf{x})$. When the number of perception variables and number of states in the extensional model is high, modeling $\rho(s_i, \mathbf{x})$ with an n-dimensional distribution $p(\mathbf{x} \mid s)$, as proposed by [103]), is extremely expensive from the computational point of view and result infeasible. A practical simplifying hypothesis can be obtained by assuming that ρ factorizes in n perception functions, one for each perception variable. This means that

$$\rho(s_i, \mathbf{x}) = \prod_{j=1}^{n} \rho_j(s_i, x_j)$$

where each $\rho_j(s_i, x)$ is an unidimensional probability density function. The additional advantage of this factorization is that it allows to associate different thresholds to each perception variable for the same state, instead of a single threshold. We therefore replace the single threshold t in Algorithm PAL with a vector $\boldsymbol{t} = (t_1, \ldots, t_n)$ where $t_i \in \mathbb{R}^+$ is the threshold associated to the i-th perception variable. The set of abstract states on which we have to maximize the likelihood in order to find the next state is thereby defined as

$$S' = \{s_i \in S \mid \rho(s_i, \mathbf{x}) \geq \boldsymbol{t}\}$$

where condition $\rho(s_i, \mathbf{x}) \geq \boldsymbol{t}$ stands for $\bigwedge_{j=1}^{n} \rho_j(s_i, x_j) \geq t_j$ (steps 4, 18, 36 of Algorithm PAL). A concrete, but still very general, model for a single variable perception function which we decided to adopt is the normal distribution. We therefore suppose that, for every state s_i and perception variable j, $\rho_j(s_i, x_j)$ is the normal distribution $\mathcal{N}(x_j \mid \mu_{ij}, \sigma_{ij})$ with mean μ_{ij} and variance σ_{ij}.

The parameters μ_{ij} and σ_{ij} can be learned online (step 55 of Algorithm PAL). Given a sequence of m observations $(\mathbf{x}^{(k)})_{k=1}^{m}$ associated to the same state s_i, the mean μ_{ij} of the j-th perception variable is updated as follows:

$$\mu_{ij} = \frac{2}{m(m+1)} \sum_{k=0}^{m-1} (m-k) x_j^{(m-k)}$$

For each perception variable, its mean in the state s_i is set to the normalized weighted sum of all perception observations associated with the state s_i. The first observation $\mathbf{x}^{(1)}$ is the one associated with the state when it is created; $\mathbf{x}^{(m)}$ is the last perception associated with the state by procedure PAL. The oldest the observation, the less weight is given. We assume that the standard deviation σ_{ij} keeps unchanged since given by the sensors, although in principle our approach could learn it from the data.

The choice of the sequence $\boldsymbol{t}$ is important since it strongly affects the agent's capability to correctly build the extensional model. The higher the thresholds the more states are created. With a very low threshold, redundant states can be introduced, i.e., states that correspond to very similar perceptions; from these states agents have to take the same decision to reach a goal state, and hence they should be clustered in the same state. On the other hand, if the thresholds are too low, then more than one abstract state is collapsed in a unique extensional state. A reasonable setting for t_i can be obtained by defining $t_i = \mathcal{N}(2\sigma_{noise,i}|0, \sigma_{noise,i})$, where $\sigma_{noise,i}$ is the maximum measurement noise of the sensor associated to the i-th perception variable.

4.4 PAL example

Example 4. *Figure 4.4 shows an example of an agent, placed in an unknown building with several rooms, which is asked to move a package between two different rooms. The agent is equipped with a position sensor* $\langle x_{gps}, y_{gps} \rangle$*, and an*

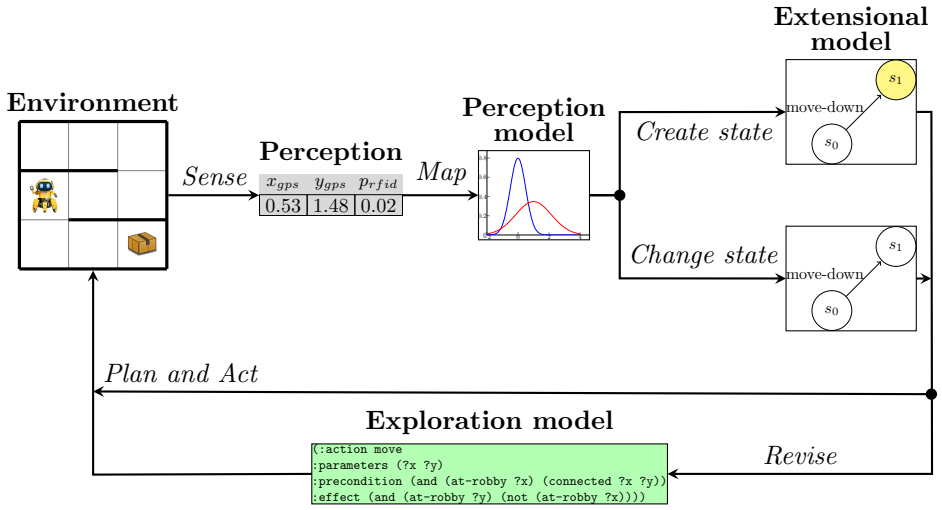


Figure 4.4: PAL example

RFID passive sensor that can receive signals from the RFID active tag p_{rfid} attached to the package. Firstly, the agent perceives the environment through its sensors. Afterward, the perception is mapped into a state of the extensional model of the planning domain by means of a perception model. Given the perception of the agent, if the likelihood of being in a state of the extensional model of the planning domain is higher than a given threshold, then the agent creates a new state associated with such a perception. Otherwise, the extensional model state that maximizes the likelihood of the perception is chosen and the agent changes state. Next, the agent plans firstly with the extensional model and then with the exploration model. If in the extensional model there is a plan for reaching a goal state, then such a plan is executed. Otherwise, if no plan can be computed in the extensional model learned so far by the agent, then the agent plans by means of the exploration model.

4.5 Experimental Analysis

In our experimental analysis, we evaluate the effectiveness of the proposed approach and, in particular, the usefulness of using the exploration planner for guiding the search. As exploration planner we used the well-known planner FastDownward [50]. All experimental tests were conducted on an Intel Xeon Skylake 2.3 GHz with 8 cores and 128 GB of RAM. The time limit for each run of PAL was 60 minutes, after which termination was forced.

4.5.1 Benchmarks and simulators

Our benchmarks derive from three well-known planning domains: Logistics, Grid, and Rovers. We assume that these domains are approximations of the world where agents act. For instance, Logistics concerns moving packages among cities by airplanes and trucks; in the real world, only a number of air routes are permitted, while in the standard PDDL Logistics domain airplanes can move between any pair of airports. This simplification could be adopted because, e.g., the exact network of the air routes is unknown to the domain model engineer.

For each of the considered domains, we developed a simulator that simulates the physics of the domain with some discrepancies w.r.t. the available PDDL model. Differently from the Logistics domain model, a number of air and road routes are forbidden in the simulator. Specifically, all the airports but one are partitioned into two sets, each airplane can visit airports in only one of the two sets, plus a special airport that is not in either set. Similarly, all the locations within each city are divided into two overlapping sets, each truck can visit locations in only one of the two sets. We assume that GPS trackers are installed on board of both trucks and airplanes, RFID readers are installed on board of trucks as well as in storage areas, and that there are RFID tags stuck on packages. The simulation of an action of the Logistics domain outputs a perception consisting of readings made by GPS trackers and RFID readers. The GPS coordinates of a location are random numbers ranging from 1500 to 30,000; each reading made by the GPS tracker of a vehicle at a certain location is a pair of GPS coordinates corresponding to the location of the vehicle with a noise ranging from 0 to 5; the reading made by the RFID reader at location l for the RFID tag of package p is a random number ranging from 0.8 to 1, if p is at l; it is a random number ranging from 0 to 0.2 otherwise; the same reading is made by the RFID reader installed on board of a vehicle for the RFID tag of a package.

Domain Grid concerns moving a robot among a grid of rooms, some of which are closed by doors that can be opened by keys located in different rooms. A robot can move from room x to room y only if the two rooms are adjacent in the grid. Differently from the (standard PDDL) Grid domain, in the simulator x and y need to be connected in order for the robot to move between the two adjacent rooms, and only 3 over 4 adjacent rooms are connected. We assume

Table 4.1: Minimum and maximum number of domain states (1st column), actions (2nd column), perception variables (3rd column), and number of states learned by PAL with the CONTINUE setting (4th column) over the instances of our benchmark domains solved by PAL.

Domain	S	A	PV	LS
Logistics	[e+21, e+219]	[650, 151400]	[269, 18152]	[856, 4864]
Grid	[e+07, e+35]	[726, 26135]	[47, 147]	[204, 1983]
Rovers	[e+10, e+68]	[362, 33732]	[165, 3961]	[114, 1286]

that a GPS tracker and an RFID reader are installed on board of the robot, keys have GPS trackers and RFID tags, and there are sensors mounted on doors that detect whether doors are open or closed. The simulation of an action of the Grid domain outputs a perception consisting of readings made by GPS trackers, RFID readers, and door sensors.

The GPS coordinates of room (x, y) in the grid are $(100 \cdot x, 100 \cdot y)$. The reading made by the GPS tracker of the robot at a certain room is a pair of GPS coordinates corresponding to the room coordinates with a noise ranging from 0 to 5; the same reading is done by the GPS trackers mounted on keys. The reading made by the RFID reader of the robot for the RFID tag of a certain key is a random number ranging from 0.8 to 1.0, if the key is grasped by the robot; it is a random number ranging from 0 to 0.2 otherwise. Similarly, the reading of the door sensors is a random number ranging from 0.8 to 1.0, if the door is open; it is a random number ranging from 0 to 0.2, if the door is closed.

Domain Rovers concerns moving rovers on the surface of a planet, taking images, collecting samples, and communicating images back to a lander. A rover at a waypoint can take an image of an objective only if the objective is visible from the waypoint. Similarly, a rover at a waypoint can communicate back data to the lander only if the lander is visible from the waypoint. Differently from the PDDL domain model, the simulator does not allow to take images at half of the waypoints from which an objective is visible, and it does not allow to communicate back data to the lander at half of the waypoints from which the lander is visible. We assume that rovers have onboard GPS trackers, and that there are sensors that output real numbers on the basis of the truth values of facts of the domain. The simulation of an action of domain Rovers outputs a perception consisting of readings made by GPS trackers and sensors.

The GPS coordinates of a waypoint are random numbers ranging from 0 to 3400; the reading made by the GPS tracker of a rover at a waypoint is the same GPS coordinates as for the waypoint with a noise ranging from 0 to 5; the reading of the sensors is a random number ranging from 0.8 to 1.0, if the fact the sensor detects is true, it is a random number ranging from 0 to 0.2 otherwise.

We generated and tested the following PAL problems: 37 problems derived from the largest instances of Logistics used in the first two International Planning Competitions (IPCs) [9, 82]; the 5 problems derived from the instances of Grid used in the first IPC [82] plus 30 problems derived from randomly generated instances; and 40 problems derived from the instances of Rovers used in the third IPC [36]. The initial and goal perceptions of the PAL problems were derived from the initial states and sets of goals of the relative IPC problems.

4.5.2 Experimental results

The first experiment we conducted is running PAL with the IPC version of the planning domain for the input exploration model, and empty models for the input extensional and perception models. Algorithm 4.3 updates the exploration model when the execution of an action a in a state fails so that, when the

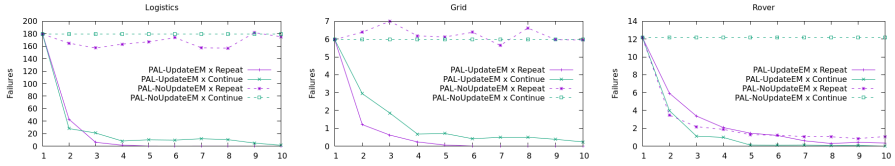


Figure 4.5: Average number of action-execution failures of PAL-NoUpdateEM and PAL-UpdateEM with settings REPEAT and CONTINUE for 10 episodes of domains Logistics, Grid, and Rovers.

exploration planner is run again from this state, the first action to execute in the new plan is different from a.

For simplicity, in the current implementation of the (automatic) revision of the exploration model, if the execution of an action fails in a state, the model is modified in a way that such an action is never executable. Even if the physics of the world encoded in the exploration model and in the simulator have discrepancies (as previously described), PAL using FastDownward can solve 30 over 37 instances of Logistics, and all the instances of Grid and Rovers. Table 4.1 shows that PAL using the exploration model is able to solve quite large problems. On the contrary, PAL without an exploration model solves no problem of our benchmarks, because it explores the world states randomly and only tiny problems where goals are "accidentally" reached can be solved.

Then, we tested PAL with non-empty input extensional and perception models. For each PAL problem, we repeatedly ran PAL with two different settings. In the first setting, PAL is run with the same initial and goal perceptions as those of the PAL problem. We call each of these run an *episode*. In the second setting, for each PAL problem, we constructed a set $\mathbf{X}_g$ of goal perceptions derived from randomly generated sets of PDDL goals. For the first episode, PAL is run with the same initial and goal perceptions as those of the PAL problem; for each other episode, PAL is run with the last perception sensed in the previous episode as initial perception and a perception among those in $\mathbf{X}_g$ as goal perception. Essentially, for this second setting PAL continues to plan for incoming goals. In the following, the first and second settings are called REPEAT and CONTINUE, respectively.

We considered ten episodes and two versions of PAL. For both versions, the input knowledge is the same but the exploration models are different. For every episode except the first one, the extensional and perception models are those derived at the end of the previous episode; for the first episode they are empty. One of the two versions of PAL has in input the IPC domain model as exploration model, the other version has in input the exploration model derived at the end of the previous episode. We denote these two versions of PAL with PAL-NoUpdateEM and PAL-UpdateEM, respectively.

Figure 4.5 shows the number of action-execution failures of PAL-NoUpdateEM and PAL-UpdateEM for settings REPEAT and CONTINUE. Since the exploration model is an approximation of the real world, the execution (through the sim-

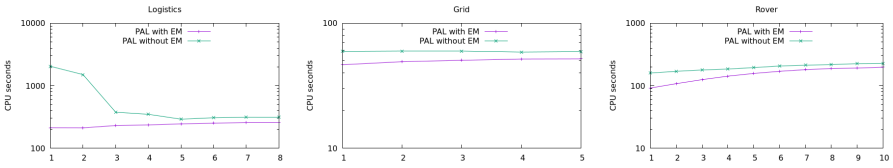


Figure 4.6: Average CPU-time of PAL using the REPEAT setting with/without the exploration model (EM) for up to 10 episodes of domains Logistics, Grid, and Rovers.

ulator) of an action in the plan computed by the exploration planner can fail. The results in Figure 4.5 show that for both settings REPEAT and CONTINUE the number of failures when the exploration model is updated is significantly lower than when no update is done among the episodes. For the REPEAT setting and every considered domain, the number of failures reduces nearly to zero at the fifth episode. Remarkably, even for the CONTINUE setting the number of failures tends to decrease and is close to zero after few episodes, showing the usefulness of the learned knowledge.

When the agent's goals are satisfied in a state previously reached (and learned) by the agent, using the extensional planner can provide great computational benefits. This is because, typically, the (learned) state space searched by the extensional planner is much smaller than the state space induced by the exploration model searched by the exploration planner. To evaluate these benefits, we conducted an experiment using the REPEAT setting. First, we run PAL using the exploration planner; then, we run PAL without using the exploration planner but having the extensional and perception models learned by PAL in the first run (that used the exploration planner). Figure 4.6 shows the average CPU time of PAL with/without the exploration planner for up to ten episodes. For all the instances and episodes greater than 8 in Logistics, and greater than 5 in Grid, we have no action-execution failure for PAL using the exploration planner; hence the performance gap is the same as for the last episode shown in the figure. As expected, the CPU time of PAL using only the extensional planner is lower than when using the exploration planner. However, achieving the goal by the extensional planner is still, somewhat surprisingly, quite expensive. This is because determining the next current state of the agent from the sensed perception can be computationally much expensive when the number of perception variables and (learned) states in the extensional model is high.

To determine the next state, we filter the set of previously reached (learned) states according to the set J of perception variables that have been significantly changed by the execution of the last action. The next state is selected from the set of states satisfying $\rho_j(x_j \mid s) > t_j$ for each perception $j \in J$ according to the max-likelihood criteria. If this set is empty, then a new state is introduced.

In Figure 4.7, such a strategy is denoted by *"State filtering"*. Finally, we consider another strategy. If the perception $\mathbf{x}$ is obtained by executing action a in state s and the extensional model contains the transition (s, a, s'), then, if

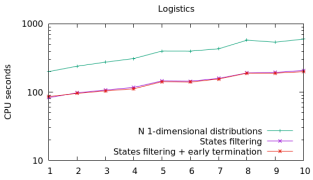


Figure 4.7: Average CPU-time of PAL with the CONTINUE setting using five different methods for determining the next search state for 10 episodes of domain Logistics.

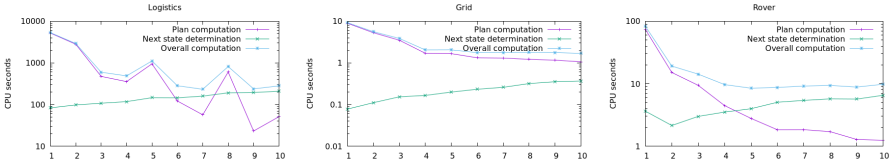


Figure 4.8: Average CPU-time for planning, determining the next search state, and total time required by PAL with the CONTINUE setting for each episode among 10 episodes of domains Logistics, Grid, and Rovers.

for each perception variable i the likelihood of sensing x_i being in s' is above threshold t_i, the next state is s'. We run PAL using such a strategy together with the strategy for filtering states; this version of PAL is denoted by *"State filtering + early termination"*. The results in Figure 4.7 show that, on average, for Logistics the filtering of states significantly improves the performance, while the speedup obtained using the early termination is negligible.

Figure 4.8 shows, for the CONTINUE setting, the average CPU time required by PAL for planning and for determining the next state w.r.t. the average total CPU time. For the last episodes, determining the next state is more time-consuming than using the exploration planner. This is because, for the last episodes, (i) the number of action-execution failures is low or zero, and hence the number of times the exploration planner is run is also low; (ii) the computational cost required to determine the next states increases with the number of visited states, which progressively increases with the episodes.

Chapter 5

Online Learning of Action Models

Several works have addressed the task of learning action models, and have provided important results from different perspectives and according to different assumptions, see, e.g., Section 3.2.

However, most of the recent and state-of-the-art methods perform learning offline, and require as input a set of plan traces generated by previously executed actions. This has two major drawbacks. First, often agents need to learn the model of the domain *online*, because they need to explore an unknown environment, acquire information, and learn a model by experimenting with the execution of their actions incrementally, step by step. This is the case of many applications in robotics, e.g., in SLAM [108], where the robot tries to build a map of the environment by exploration, or in the Robocup Rescue [60], where the robot needs to explore the environment to perform a rescue task. Second, previous work on learning action models does not deal with the problem of generating informative plan traces. As stated in the conclusions of [4], generating informative plan traces for learning planning action models is still an open issue. Indeed, if the available set of plan traces does not contain informative examples, there is little chance to learn all action preconditions, since some preconditions can be only discovered by specific plans that can unlikely be generated randomly [35].

We propose a new approach that does not suffer these drawbacks, focusing on the case of learning STRIPS action schema expressed in PDDL, and under the assumption of full observability of the states reached by the agent. We propose an algorithm, called OLAM algorithm (Online Learning of Action Models), for learning action models online, incrementally during the execution of plans. A key aspect of OLAM is that it combines and interleaves the activity of learning action preconditions and effects with an exploration phase that selects which plan to execute. In this way, OLAM generates plan traces to reach certain goal states, decided online, which are useful for the learning task.

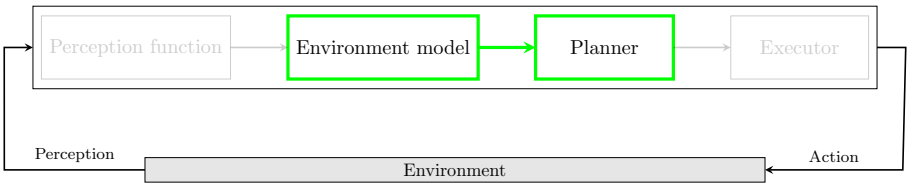


Figure 5.1: OLAM within the agent environment interface.

Beyond proving termination, we analyze our algorithm to show some important theoretical properties that are defined according to the state transitions of the models learned by the algorithm. In particular, we prove that OLAM is correct, i.e., it learns action models which generate only the state transitions generated by the planning domain modeling the true environment where the agent acts. Moreover, OLAM is "integral", i.e., it learns action models that generate all the transitions of the true environment with respect to the states that can be reached by the algorithm.

We also provide substantial empirical evidence of the good learning performance of OLAM using a large set of benchmarks from the International Planning Competitions (IPCs). Finally, we experimentally compare OLAM with a recent and state-of-the-art method for learning action models offline, showing that online learning can be much more effective.

With respect to the agent-environment interface shown in Figure 1.2, OLAM focuses on: *(i)* learning the environment module, since it learns the action preconditions and effects which describe the environment dynamic; *(ii)* the planner module, since OLAM plans to reach informative states from which it can learn by attempting to execute an action. In particular, there is no focus on the perception function module, since OLAM directly perceives the world through symbolic observations, which are assumed to be correct and complete. Finally, there is the simplifying assumption that symbolic actions can be executed by the agent, and that there is feedback about their successful/unsuccessful execution.

5.1 Action Model Learning Problem

Assuming that the sets $\mathcal{P}$, $\mathcal{O}$ and $\mathcal{C}$ are known by the agent, its task is to *learn a planning domain by executing the actions available in $\mathcal{O}$ over constants in $\mathcal{C}$, observing, and determining what are their preconditions and effects on the environment described in terms of the properties in $\mathcal{P}$. In formal terms, the agent has to build an action model $\mathcal{M} = \langle \mathcal{P}, \mathcal{O}, \mathcal{H} \rangle$, i.e., the preconditions and effects of every action schema in the domain of $\mathcal{H}$.* We assume that the dynamics of the environment where the agent acts, which is unknown by the agent, is fully described by the finite state machine $\mathcal{M}'(\mathcal{C})$, where $\mathcal{M}' = \langle \mathcal{P}, \mathcal{O}, \mathcal{H}' \rangle$ is an action model called *Ground-Truth Model* (GTM).

The following definitions state the notions of correctness and integrity for the learned planning domain $\mathcal{M} = \langle P, \mathcal{O}, \mathcal{H} \rangle$ w.r.t. the GTM.

Definition 12 (Correctness)**.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be two action models and $\mathcal{M}(\mathcal{C}) = \langle \mathcal{S}, \mathcal{A}, \delta \rangle$ and $\mathcal{M}'(\mathcal{C}) = \langle \mathcal{S}, \mathcal{A}, \delta' \rangle$ be their FSMs with respect to a set of constants $\mathcal{C}$. We say that*

1. *$\mathcal{M}(\mathcal{C})$* correctly approximates *$\mathcal{M}'(\mathcal{C})$ from a state $s_0 \in \mathcal{S}$ if, for every state s_n reachable from s_0 in $\mathcal{M}(\mathcal{C})$, $\langle s_n, a, s \rangle \in \delta$ implies $\langle s_n, a, s' \rangle \in \delta'$ for some $s' \supseteq s$.*

2. *$\mathcal{M}(\mathcal{C})$* correctly approximates *$\mathcal{M}'(\mathcal{C})$ if $\mathcal{M}(\mathcal{C})$ correctly approximates $\mathcal{M}'(\mathcal{C})$ from every state in $\mathcal{S}$;*

3. *$\mathcal{M}$* correctly approximates *$\mathcal{M}'$ if $\mathcal{M}(\mathcal{C})$ correctly approximates $\mathcal{M}'(\mathcal{C})$ for every set of constants $\mathcal{C}$.*

A plan is *valid* when the actions in the plan are "executable" and the plan achieves a given set of (positive) goals. Therefore, when the learned model correctly approximates the GTM, any valid plan computed by using the learned model is also valid for the GTM.

Definition 13 (Integrity)**.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be two action models and $\mathcal{M}(\mathcal{C}) = \langle \mathcal{S}, \mathcal{A}, \delta \rangle$ and $\mathcal{M}'(\mathcal{C}) = \langle \mathcal{S}, \mathcal{A}, \delta' \rangle$ be their FSMs with respect to a set of constants $\mathcal{C}$. We say that*

1. *$\mathcal{M}(\mathcal{C})$* integrally approximates *$\mathcal{M}'(\mathcal{C})$ from a state $s_0 \in \mathcal{S}$ if, for every state s_n reachable from s_0 in $\mathcal{M}(\mathcal{C})$, $\langle s_n, a, s' \rangle \in \delta'$ implies $\langle s_n, a, s \rangle \in \delta$ for some $s \supseteq s'$;*

2. *$\mathcal{M}(\mathcal{C})$* integrally approximates *$\mathcal{M}'(\mathcal{C})$ if $\mathcal{M}(\mathcal{C})$ integrally approximates $\mathcal{M}'(\mathcal{C})$ from every state in $\mathcal{S}$;*

3. *$\mathcal{M}$* integrally approximates *$\mathcal{M}'$ if $\mathcal{M}(\mathcal{C})$ integrally approximates $\mathcal{M}'(\mathcal{C})$ for every set of constants $\mathcal{C}$.*

Therefore, when the learned model integrally approximates the GTM, any valid plan for the GTM is also a valid plan for the learned model.

5.2 OLAM Algorithm

In the proposed approach, the agent constructs and executes informative plan traces for learning the planning domain. Algorithm 2 shows the pseudocode of the OLAM (*Online Learning of Action Models*). The input of the algorithm is the same sets of predicates and operator names (with their associated arity) of the GTM, and a set $\mathcal{C}$ of constants representing the objects of the environment explored by the agent. OLAM produces in the output two planning domains $\mathcal{M}$ and $\mathcal{M}_?^-$. The former is such that $\mathcal{M}(\mathcal{C})$ correctly and integrally approximates $\mathcal{M}'(\mathcal{C})$ from the state of the environment when OLAM terminates. The latter correctly approximates $\mathcal{M}'$.

We adopt the following notations. Let $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$ and $\boldsymbol{c} = \langle c_1, \ldots, c_n \rangle$ two n-tuple of distinct parameters and constants. If p is an m-ary predicate, $p(\boldsymbol{x})$

denotes an atom $p(x_{i_1}, \ldots, x_{i_m})$ for some m-tuple of indexes $1 \leq i_1, \ldots, i_m \leq n$; and $p(\boldsymbol{c})$ the atom obtained by replacing x_i with c_i in $p(\boldsymbol{x})$. In the following, the indexing will be left implicit. OLAM incrementally builds the following sets:

1. $\mathsf{pre}(op)$, which contains the preconditions of the operator op; it is initialized to the whole set of lifted atoms (line 2); an atom $p(\boldsymbol{x})$ is removed from $\mathsf{pre}(op)$ whenever an instance $op(\boldsymbol{c})$ of op is executed successfully in a state s and $p(\boldsymbol{c}) \notin s$ (line 19).

2. $\mathsf{eff}^+_!(op)$ and $\mathsf{eff}^-_!(op)$, which contains the set of lifted positive and negative effects of op learned by the agent; they are initially empty (line 3); a lifted atom $p(\boldsymbol{x})$ is added to $\mathsf{eff}^+_!(op)$ (resp. $\mathsf{eff}^-_!(op)$) if the execution of an instance $op(\boldsymbol{c})$ of op in state s makes $p(\boldsymbol{c})$ become true (resp. false) in the resulting state (lines 20-21).

3. $\mathsf{eff}^+_?(op)$ and $\mathsf{eff}^-_?(op)$, which are sets of lifted atoms that could become part of the positive or negative effects of op; they are initialized to the entire set of lifted atoms (line 2); a lifted atom $p(\boldsymbol{x})$ is removed from $\mathsf{eff}^+_?(op)$ (resp. $\mathsf{eff}^-_?(op)$) if $p(\boldsymbol{x})$ is discovered to be a positive or negative effect or if the atom $p(\boldsymbol{c})$ is false (resp. true) in a state s and remains false (resp. true) after executing successfully an instance $op(\boldsymbol{c})$ of op in s (lines 22-23).

4. $\mathsf{pre}_\perp(op)$, which is a set of sets of lifted preconditions for op such that in every non-empty set in $\mathsf{pre}_\perp(op)$ there is at least one precondition of op; $\mathsf{pre}_\perp(op)$ is initialized to a set including only the empty set (line 4); $\mathsf{pre}_\perp(op)$ is augmented by the set formed by any lifted fact $p(\boldsymbol{x})$ such that $p(\boldsymbol{c})$ is false in a state s, if the execution of an instance $op(\boldsymbol{c})$ of op fails in s (line 26).

5. $\mathsf{eff}^+_{!?}(op)$ and $\mathsf{eff}^-_{!?}(op)$, which are derived sets denoting $\mathsf{eff}^+_!(op) \cup \mathsf{eff}^+_?(op)$ and $\mathsf{eff}^-_!(op) \cup \mathsf{eff}^-_?(op)$.

Let denote the sets of preconditions and positive/negative effects of any operator op of $\mathcal{M}'$ by $\mathsf{pre}'(op)$ and $\mathsf{eff}'^{+/-}(op)$, respectively. The update of the sets built by OLAM guarantees that $\mathsf{pre}(op)$ is a superset of $\mathsf{pre}'(op)$, $\mathsf{eff}^{+/-}_!(op)$ are subsets of $\mathsf{eff}'^{+/-}(op)$, and $\mathsf{eff}^{+/-}_{!?}(op)$ are superset of $\mathsf{eff}'^{+/-}_{!?}(op)$.

At each iteration of the external loop (lines 7–31), the agent selects a state s' and a ground action $op'(\boldsymbol{c}')$. s' is reachable from the current state with the model $\mathcal{M}$ learned so far; the ground action $op'(\boldsymbol{c}')$ is such that its execution in s' could provide to the agent some additional information about the preconditions, the positive, or the negative effects of op'. This condition is formalised by (5.1)–(5.3). In particular, if condition (5.1) holds, the preconditions of op' could be refined by executing $op'(\boldsymbol{c}')$ in s', because s' does not contain all the preconditions of $op'(\boldsymbol{c}')$. Indeed if $op'(\boldsymbol{c}')$ will succeed, then the preconditions which are false in s' can be eliminated. If condition (5.2) (resp. (5.3)) holds, some positive (resp. negative) effects of op' could be learned, because $op'(\boldsymbol{c}')$

Algorithm 2 OLAM

Input: $\mathcal{M} = \langle \mathcal{P}, \mathcal{O}, \{\mathsf{par}(op), \emptyset, \emptyset\}_{op \in \mathcal{O}} \rangle$, $\mathcal{C}$.

1: $s \leftarrow$ OBSERVE()
2: $\forall op \in \mathcal{O}, \mathsf{eff}^-_?(op) \leftarrow \mathsf{eff}^+_?(op) \leftarrow \mathsf{pre}(op) \leftarrow \mathcal{P}(\mathsf{par}(op))$
3: $\forall op \in \mathcal{O}$, $\mathsf{eff}^-_!(op) \leftarrow \mathsf{eff}^+_!(op) \leftarrow \emptyset$
4: $\forall op \in \mathcal{O}$, $\mathsf{pre}_\perp(op) \leftarrow \{\emptyset\}$
5: $\mathcal{M} \leftarrow \langle \mathcal{P}, \mathcal{O}, \{\mathsf{par}(op), \mathsf{pre}(op), \mathsf{eff}^+_!(op), \mathsf{eff}^-_!\}_{op \in \mathcal{O}} \rangle$
6: $\pi \leftarrow nil$
7: **while** $\exists s', op'(\boldsymbol{c}')$ such that s' is reachable from s by $\mathcal{M}(\mathcal{C})$ and (5.1)$\vee$(5.2)$\vee$(5.3) holds **do**
8: $\quad \pi \leftarrow$ PLAN$(\mathcal{M}(\mathcal{C}), s, s')$
9: $\quad$ **while** $\pi \neq nil$ **do**
10: $\quad\quad$ **if** $\pi \neq \langle\rangle$ **then**
11: $\quad\quad\quad op(\boldsymbol{c}) \leftarrow$ POP(π)
12: $\quad\quad$ **else**
13: $\quad\quad\quad op(\boldsymbol{c}) \leftarrow op'(\boldsymbol{c}')$
14: $\quad\quad\quad \pi \leftarrow nil$
15: $\quad\quad$ **end if**
16: $\quad\quad \boldsymbol{x} \leftarrow \mathsf{par}(op)$
17: $\quad\quad$ **if** EXECUTE$(op(\boldsymbol{c}))$ does not fail **then**
18: $\quad\quad\quad s_{next} \leftarrow$ OBSERVE()
19: $\quad\quad\quad \mathsf{pre}(op) \leftarrow \{p(\boldsymbol{x}) \in \mathsf{pre}(op) \mid p(\boldsymbol{c}) \in s\}$
20: $\quad\quad\quad \mathsf{eff}^+_!(op) \leftarrow \mathsf{eff}^+_!(op) \cup \{p(\boldsymbol{x}) \mid p(\boldsymbol{c}) \in s_{next} \setminus s\}$
21: $\quad\quad\quad \mathsf{eff}^-_!(op) \leftarrow \mathsf{eff}^-_!(op) \cup \{p(\boldsymbol{x}) \mid p(\boldsymbol{c}) \in s \setminus s_{next}\}$
22: $\quad\quad\quad \mathsf{eff}^+_?(op) \leftarrow \mathsf{eff}^+_?(op) \setminus \{p(\boldsymbol{x}) \mid p(\boldsymbol{c}) \notin s \cap s_{next}\}$
23: $\quad\quad\quad \mathsf{eff}^-_?(op) \leftarrow \mathsf{eff}^-_?(op) \setminus \{p(\boldsymbol{x}) \mid p(\boldsymbol{c}) \in s \cup s_{next}\}$
24: $\quad\quad\quad s \leftarrow s_{next}$
25: $\quad\quad$ **else**
26: $\quad\quad\quad \mathsf{pre}_\perp(op) \leftarrow \mathsf{pre}_\perp(op) \cup \{\{p(\boldsymbol{x}) \in \mathsf{pre}(op) \mid p(\boldsymbol{c}) \notin s\}\}$
27: $\quad\quad\quad \pi \leftarrow nil$
28: $\quad\quad$ **end if**
29: $\quad\quad \mathcal{M} \leftarrow \langle \mathcal{P}, \mathcal{O}, \{\mathsf{par}(op), \mathsf{pre}(op), \mathsf{eff}^+_!(op), \mathsf{eff}^-_!\}_{op \in \mathcal{O}} \rangle$
30: $\quad$ **end while**
31: **end while**
32: $\mathcal{M}^-_? \leftarrow \langle \mathcal{P}, \mathcal{O}, \{\mathsf{par}(op), \mathsf{pre}(op), \mathsf{eff}^+_!(op), \mathsf{eff}^-_{!?}\}_{op \in \mathcal{O}} \rangle$
33: **return** $\mathcal{M}, \mathcal{M}^-_?$

Conditions in line 7:

$$\mathsf{pre}(op'(\boldsymbol{c}')) \setminus s' \notin \mathsf{pre}_\perp(op'(\boldsymbol{c}')) \tag{5.1}$$

$$\mathsf{pre}(op'(\boldsymbol{c}')) \subseteq s' \text{ and } \mathsf{eff}^+_?(op'(\boldsymbol{c}')) \not\subseteq s' \tag{5.2}$$

$$\mathsf{pre}(op'(\boldsymbol{c}')) \subseteq s' \text{ and } \mathsf{eff}^-_?(op'(\boldsymbol{c}')) \cap s' \neq \emptyset \tag{5.3}$$

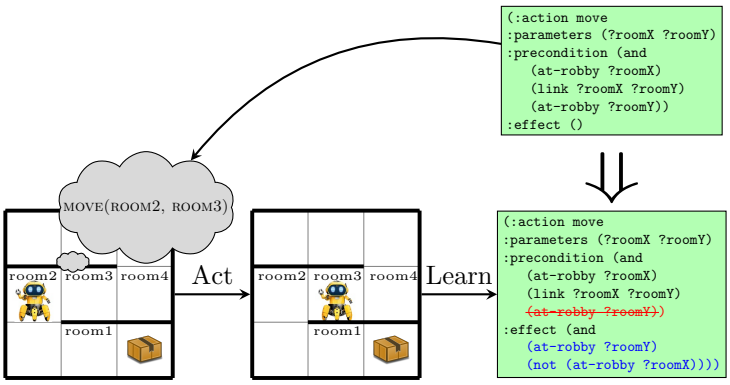


Figure 5.2: Successful action execution.

is executable in s' and s' does not contain all the facts in $\mathsf{eff}^+_?(op'(\boldsymbol{c}'))$ (resp. contains at least a fact in $\mathsf{eff}^-_?(op'(\boldsymbol{c}'))$). Indeed, the facts in $\mathsf{eff}^+_?(op'(\boldsymbol{c}'))$ but not in s' which becomes true can be added to the positive effects. Similarly, the facts that are in $\mathsf{eff}^-_?(op'(\boldsymbol{c}'))$ and in s' which becomes false can be added to the negative effects. The selection of such a state s' and action $op'(\boldsymbol{c}')$ is done by constructing a plan from the current state s to a state s' which satisfies conditions (5.1)–(5.3) for an $op'(\boldsymbol{c}')$ (line 8). If there is more than one action $op'(\boldsymbol{c}')$ that satisfies conditions (5.1)–(5.3) in s', one of them is randomly selected. The choice of s', $op'(c')$ and the associated plan is obtained by invoking PLAN with the following goal:

$$G = \bigvee_{\substack{op(\boldsymbol{c})\in\mathcal{A} \\ P^+P^-E^+E^- \text{satisfy } (i\text{–}vi)}} \left(\bigwedge_{p(\boldsymbol{c})\in P^+\cup E^-} p(\boldsymbol{c}) \wedge \bigwedge_{p(\boldsymbol{c})\in P^-\cup E^+} \neg p(\boldsymbol{c}) \right) \tag{5.4}$$

$$\begin{array}{ll} (i)\ P^-\cup E^+\cup E^-\neq\emptyset, & (ii)\ P^+\cap P^-=\emptyset, \\ (iii)\ P^+\cup P^- = \mathsf{pre}(op(\boldsymbol{c})), & (iv)\ P^-\notin \mathsf{pre}_\perp(op(\boldsymbol{c}))\setminus\{\emptyset\}, \\ (v)\ E^+\subseteq \mathsf{eff}^+_?(op(\boldsymbol{c})), & (vi)\ E^-\subseteq \mathsf{eff}^-_?(op(\boldsymbol{c})). \end{array}$$

Each disjunct in (5.4) describes a set of states from which the agent can potentially learn something by executing $op(\boldsymbol{c})$. P^+ and P^- partition the preconditions of $op(\boldsymbol{c})$ so that the atoms in P^+ are true in s', the atoms in P^- are false in s', and set P^- has not already been checked to be necessary for successfully executing $op(\boldsymbol{c})$. E^+ is a subset of possible positive effects of $op(\boldsymbol{c})$ which are false in s' and can become true by executing $op(\boldsymbol{c})$; similarly for E^-. Notice that for every state s' that contains P^+ and E^- and does not contain P^- and E^+, and every action $op'(\boldsymbol{c}')$ such that (iv) and (v) and (vi) hold, when condition (5.1) is satisfied by s' and $op'(\boldsymbol{c}')$, P^- is not empty; when condition (5.2) is satisfied by s' and $op'(\boldsymbol{c}')$, E^+ is not empty; finally, when condition (5.3) is satisfied by s' and $op'(\boldsymbol{c}')$, E^- is not empty.

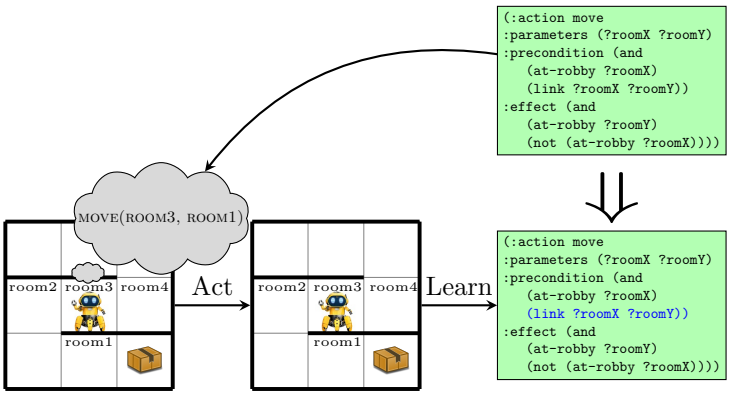


Figure 5.3: Example of action execution failure.

In the internal loop (lines 9–30), OLAM executes π and if it manages to successfully complete the execution of π (i.e., $\pi = \langle\rangle$, line 10) the ground action $op'(\boldsymbol{c}')$ will be executed in the environment where the agent acts (line 17). The dynamics of such an environment is unknown by the agent, and it determines the result returned by EXECUTE($op(\boldsymbol{c})$). When a ground action $op(\boldsymbol{c})$ is successfully executed, OLAM observes the state of the environment s_{next} resulting from the execution (line 18), and updates sets $\mathsf{pre}(op)$, $\mathsf{eff}_!^{+/-}(op)$ and $\mathsf{eff}_?^{+/-}(op)$ according to the criteria defined above (lines 19–23), an example of successful action execution is shown in Figure 5.2. If the $op(\boldsymbol{c})$ execution fails in the environment, $\mathsf{pre}_\perp(op)$ is extended as described above, and π is reset to *nil* since its execution deviates from the expected trajectory computed according to the domain $\mathcal{M}$ learned so far (lines 26-27). An example of action execution failure is shown in Figure 5.3.

5.3 OLAM Example

Figure 5.4 shows an iteration application example of the OLAM algorithm. At the beginning, the agent observes the current status of the environment, i.e. that there is a set of linked rooms, the agent is in room4 and the package in room9. Next, the agent computes a goal that can be focused on learning new effects, removing preconditions or confirming them, for learning the action model of the "move" operator. Suppose that the operator "move" has two uncertain preconditions, i.e. "(link ?roomX ?roomY)" and "(link ?roomY ?roomX)", then the goal specified in Figure 5.4 identifies the goal states where exactly one of these preconditions is false. Indeed, in such goal states, the agent can try to execute an action of type "move" and check whether these preconditions are necessary or not. Afterward, the agent plans to reach a goal state and test some preconditions or effects. Typically, at the beginning, the agent's initial state is

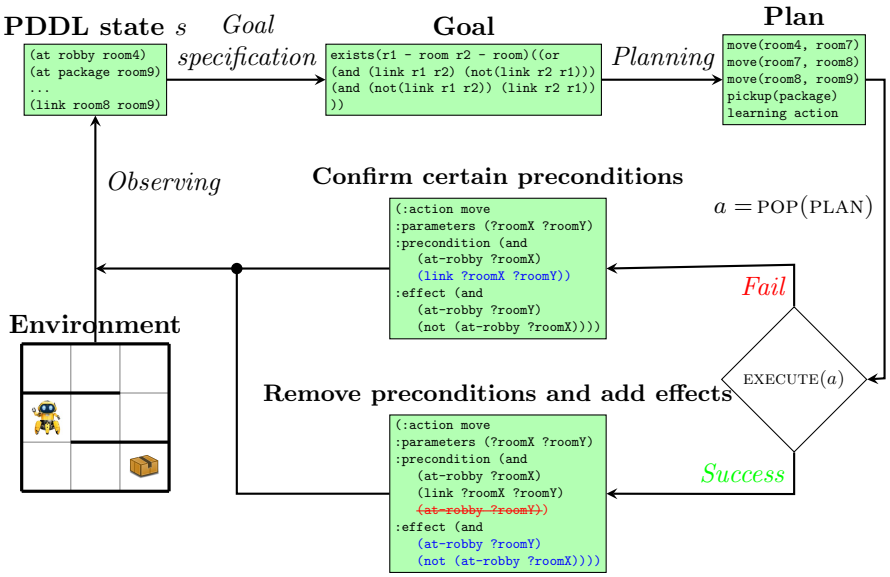


Figure 5.4: OLAM Example.

already a goal one, hence the plan contains only the action whose preconditions or effects could be learned, denoted as the "learning action" in Figure 5.4. Whenever the agent executes a "learning action", if the execution fails, then it confirms some action preconditions, since there is at least one real precondition that is not satisfied in the current state. Otherwise, if the execution succeeds, the agent can remove action preconditions that are not satisfied and learn new effects by comparing the destination state with the previous one. Finally, the agent again observes the environment status and so on, until no more learning states can be reached.

5.4 Termination, Correctness, and Integrity

Given an n-ary operator, we assume that it can be grounded only with n *different* constants. This assumption can be done without loss of generality, at the price of introducing additional operators with only one parameter in place of the set of parameters that can be grounded with the same constant. We also suppose that OLAM is run in the environment $\mathcal{M}'(\mathcal{C})$ where $\mathcal{C}$ is a set of at least $\max_{op \in \mathcal{O}} |\mathsf{par}(op)|$ constants.

In the following, the transitions relations of $\mathcal{M}$, $\mathcal{M}'$, and $\mathcal{M}_?^-$ are denoted by δ, δ', and $\delta_?^-$. Moreover, the sets of preconditions and positive/negative effects of any operator op of $\mathcal{M}'$ are denoted by $\mathsf{pre}'(op)$ and $\mathsf{eff}'^{+/-}(op)$, respectively.

Lemma 1. *For every n-ary operator op with parameters $\mathsf{par}(op) = \boldsymbol{x} = (x_1, \ldots, x_n)$, every m-ary predicate p, and every n-tuple of distinct constants $\boldsymbol{c} = (c_1, \ldots, c_n)$*

in $\mathcal{C}$*:*

1. $p(\boldsymbol{c}') \in \mathsf{pre}(op(\boldsymbol{c}))$ *iff* $p(\boldsymbol{x}') \in \mathsf{pre}(op)$
2. $p(\boldsymbol{c}') \in \mathsf{eff}^+(op(\boldsymbol{c}))$ *iff* $p(\boldsymbol{x}') \in \mathsf{eff}^+(op)$
3. $p(\boldsymbol{c}') \in \mathsf{eff}^-(op(\boldsymbol{c}))$ *iff* $p(\boldsymbol{x}') \in \mathsf{eff}^-(op)$

where $\boldsymbol{c}' = (c_{i_1}, \ldots, c_{i_m})$ *and* $\boldsymbol{x}' = (x_{i_1}, \ldots, x_{i_m})$ *with* $1 \leq i_j \leq n$ *for* $1 \leq j \leq m$*.*

Proof. Let us consider case 1 (similar proofs can be derived for cases 2 and 3). Since $\mathsf{pre}(op)$ cannot contain constants, the only way to obtain $p(\boldsymbol{c}')$ in $\mathsf{pre}(op(\boldsymbol{c}))$ is by grounding some precondition $p(\boldsymbol{x}') \in \mathsf{pre}(op)$ with $\boldsymbol{c}'$. Since we require that every parameter in $\mathsf{par}(op)$ is instantiated with a different constant, the only way to obtain $p(c_{i_1}, \ldots, c_{i_m})$ is when $\boldsymbol{x}' = (x_{i_1}, \ldots, x_{i_m})$. The opposite direction derives by the fact that if $p(\boldsymbol{x}') \in \mathsf{pre}(op)$, by grounding op's parameters in $p(\boldsymbol{x}')$ with $\boldsymbol{c}'$, we obtain $p(\boldsymbol{c}') \in \mathsf{pre}(op(\boldsymbol{c}))$. □

Lemma 2. *At every execution step of* OLAM $\mathsf{pre}(op) \supseteq \mathsf{pre}'(op)$*.*

Proof. In OLAM, $\mathsf{pre}(op)$ is initialized by $\mathcal{P}(\mathsf{par}(op))$, i.e., all the possible preconditions on the parameters of op (line 2). Then, a precondition $p(\boldsymbol{x})$ is removed from $\mathsf{pre}(op)$ when an action $op(\boldsymbol{c})$ is executed with success in s and $p(\boldsymbol{c}) \notin s$. (line 19). This implies that $p(\boldsymbol{c}) \notin \mathsf{pre}'(op(\boldsymbol{c}))$. By lemma 1 we have that $p(\boldsymbol{x}) \notin \mathsf{pre}'(op)$. This guarantees that at every execution step of the algorithm $\mathsf{pre}(op) \supseteq \mathsf{pre}'(op)$. □

Lemma 3. *At every execution step of* OLAM*, if* $\mathsf{pre}(op(\boldsymbol{c})) \subseteq$ OBSERVE()*, then* EXECUTE$(op(\boldsymbol{c}))$ *does not fail.*

Proof. Let $s =$ OBSERVE() be the result of the observation of the environment at some execution step of OLAM, i.e., the current state according to $\mathcal{M}'(\mathcal{C})$. By Lemmas 1-2, if $\mathsf{pre}(op(\boldsymbol{c})) \subseteq s$ then $\mathsf{pre}'(op(\boldsymbol{c})) \subseteq s$, which guarantees that action $op(\boldsymbol{c})$ can be executed with success from the current state according to $\mathcal{M}'(\mathcal{C})$. □

Theorem 1 (Termination)**.** *Algorithm* OLAM *terminates.*

Proof. First of all, notice that for every operator op the following properties hold:

- The size of $\mathcal{P}(\mathsf{par}(op))$ and $2^{\mathcal{P}(\mathsf{par}(op))}$ are finite and therefore $\mathsf{pre}(op)$, $\mathsf{eff}^+_?(op)$, $\mathsf{eff}^-_?(op)$ are initialized to finite sets, $\mathsf{eff}^+_!(op)$, $\mathsf{eff}^-_!(op)$ cannot be larger than $\mathcal{P}(\mathsf{par}(op))$, and $\mathsf{pre}_\perp(op)$ cannot be larger then the size of $2^{\mathcal{P}(\mathsf{par}(op))}$.
- The internal loop (lines 9–30) always terminates because at every iteration either the size of the the plan reduces by 1 unit, or the plan is set to *nil*, and if the size is 0 the plan is set to *nil*.

Given the above points, to show termination, we have to prove that at every iteration of the external loop (7–31) one of the following facts holds for some operator op:

(a) the size of $\mathsf{pre}(op)$, $\mathsf{eff}^+_?(op)$, or $\mathsf{eff}^-_?(op)$ is reduced;

(b) the size of $\mathsf{pre}_\perp(op)$, $\mathsf{eff}^+_!(op)$, or $\mathsf{eff}^-_!(op)$ is increased.

At each iteration of the external loop (lines 7–31) OLAM selects an action $op'(\boldsymbol{c}')$ and a state s' reachable from the current state s with $\mathcal{M}$ that satisfy condition ((5.1)), (5.2) or (5.3). It produces a plan $\pi = (a_1, \ldots, a_k)$ from the current state to s', and it executes the plan in the internal loop (lines 9–30). We consider separately the case where (i) π is successfully executed and the state s' is achieved, and (ii) the execution of π fails or the reached state is different from s'.

(i) *π successfully achieves s':* after k iterations of the internal loop, plan π becomes empty, and the condition at line 10 becomes true. Then, $op'(\boldsymbol{c}')$ is executed in s' (line 11). If $op'(\boldsymbol{c}')$ is executed successfully, then since $op'(\boldsymbol{c}')$ and s' satisfies at least one of the three conditions ((5.1), (5.2) and (5.3)), the following applies. If (5.1) holds then $\mathsf{pre}(op(\boldsymbol{c})) \setminus s' \neq \emptyset$ and therefore $\mathsf{pre}(op')$ is reduced (line 19); if (5.2) holds then $\mathsf{eff}^+_?(op)$ is reduced (line 22); if (5.3) holds then $\mathsf{eff}^-_?(op)$ is reduced (line 23). If $op'(\boldsymbol{c}')$ fails in the state s' (line 25), then by Lemma 3 it means that $\mathsf{pre}(op'(\boldsymbol{c}')) \not\subseteq s'$, and therefore conditions ((5.2)) and ((5.3)) are false, which implies that condition ((5.1)) is true. This guarantees that at line 26 $\mathsf{pre}_\perp(op)$ is extended.

(ii) *π fails to achieve s':* Since plan π is computed according to $\mathcal{M}$ and, by Lemmas 1-2, for any action $op(\boldsymbol{c})$ the set of preconditions of $op(\boldsymbol{c})$ in $\mathcal{M}(\mathcal{C})$ contains the preconditions of $op(\boldsymbol{c})$ in $\mathcal{M}'(\mathcal{C})$, then the failure of π implies that after $j \leq k$ iterations of the internal loop, the observed state (computed by executing $a_1, \ldots, a_j$ from s in $\mathcal{M}'(\mathcal{C})$) is different from the state computed by executing $a_1, \ldots, a_j$ from s in $\mathcal{M}$. Let i be the smallest of such a j, and $a_i = op_i(\boldsymbol{c}_i)$. Then $(s_{i-1}, op_i(\boldsymbol{c}_i), s_i) \in \delta$ and $(s_{i-1}, op_i(\boldsymbol{c}_i), s'_i) \in \delta'$ with $s'_i \neq s_i$. Since $a_{i-1} = op_{i-1}(\boldsymbol{c}_{i-1})$ modifies only the atoms containing the constants $\boldsymbol{c}_{i-1}$ contained in s_i and s'_i differ on some $p(\boldsymbol{c}_{i-1})$. If $p(\boldsymbol{c}_{i-1}) \in s'_i \setminus s_i$ then, at the $i-1$-th iteration of the internal loop, $\mathsf{eff}^+_!(op_{i-1})$ is extended (line 20); if $p(\boldsymbol{c}_{i-1}) \in s_i \setminus s'_i$, then $\mathsf{eff}^-_!(op_{i-1})$ is extended (line 21).

□

Lemma 4. *At every execution step of OLAM,* $\mathsf{eff}^{+/-}_!(op) \subseteq \mathsf{eff}'^{+/-}(op)$.

Proof. In OLAM, $\mathsf{eff}^{+/-}_!(op)$ are initialized by the empty set (line 3), and therefore $\mathsf{eff}^{+/-}_!(op) \subseteq \mathsf{eff}'^{+/-}(op)$ initially holds. Suppose that at some point $p(\boldsymbol{x}) \in \mathsf{eff}^+_!(op)$. Then, there exists a state s and an action $op(\boldsymbol{c})$ such that

the execution of $op(\boldsymbol{c})$ from s adds $p(\boldsymbol{x})$ to $\mathsf{eff}^+_!(op)$, which implies $p(\boldsymbol{c}) \notin s$ and $p(\boldsymbol{c}) \in s_{next}$ (line 20), where s_{next} is the state resulting from the execution of $op(\boldsymbol{c})$ in $\mathcal{M}'(\mathcal{C})$. Since $s_{next} \subseteq s \cup \mathsf{eff}'^+(op(\boldsymbol{c}))$, then $p(\boldsymbol{c}) \in \mathsf{eff}'^+(op(\boldsymbol{c}))$. By Lemma 1, we have that $p(\boldsymbol{x}) \in \mathsf{eff}'^+(op)$. A similar proof can be done to show that $\mathsf{eff}^-_!(op) \subseteq \mathsf{eff}'^-(op)$. □

Lemma 5. *At every execution step of OLAM,* $\mathsf{eff}^{+/-}_{!?}(op) \supseteq \mathsf{eff}'^{+/-}(op)$.

Proof. In OLAM, $\mathsf{eff}^-_?(op)$ is initialized by $\mathcal{P}(\mathsf{par}(op))$ (line 2), and therefore $\mathsf{eff}^-_{!?}(op) \supseteq \mathsf{eff}^-_? \supseteq \mathsf{eff}'^-(op)$ initially holds. Suppose that at some point $p(\boldsymbol{x}) \notin \mathsf{eff}^-_{!?}(op)$. This means that there is a state s and an action $op(\boldsymbol{c})$ such that the execution of $op(\boldsymbol{c})$ from s removes $p(\boldsymbol{x})$ from $\mathsf{eff}^-_?(op)$, which implies that $p(\boldsymbol{c}) \in s \cup s_{next}$ (line 21), where $s_{next} = s \cup \mathsf{eff}'^+(op(\boldsymbol{c})) \setminus \mathsf{eff}'^-(op(\boldsymbol{c}))$ is the state resulting from the execution of $op(\boldsymbol{c})$ in $\mathcal{M}'(\mathcal{C})$. If $p(\boldsymbol{c}) \notin s_{next}$ then $p(\boldsymbol{c}) \in s$ and this implies that $p(\boldsymbol{x})$ is added to $\mathsf{eff}^-_!(op)$, which contradicts the fact that $p(\boldsymbol{x}) \notin \mathsf{eff}^-_{!?}(op)$; therefore we have that $p(\boldsymbol{c}) \in s_{next}$, which implies that $p(\boldsymbol{x}) \notin \mathsf{eff}'^-(op)$, as required.

In OLAM, $\mathsf{eff}^+_?(op)$ is initialized by $\mathcal{P}(\mathsf{par}(op))$ (line 2), and therefore $\mathsf{eff}^+_{!?}(op) \supseteq \mathsf{eff}^+_? \supseteq \mathsf{eff}'^+(op)$ initially holds. Suppose that at some point $p(\boldsymbol{x}) \notin \mathsf{eff}^+_{!?}(op)$. This means that there is a state s and an action $op(\boldsymbol{c})$ such that the execution of $op(\boldsymbol{c})$ from s removes $p(\boldsymbol{x})$ from $\mathsf{eff}^+_?(op)$, which implies that $p(\boldsymbol{c}) \notin s \cap s_{next}$ (line 22), where $s_{next} = s \cup \mathsf{eff}'^+(op(\boldsymbol{c})) \setminus \mathsf{eff}'^-(op(\boldsymbol{c}))$ is the state resulting from the execution of $op(\boldsymbol{c})$ in $\mathcal{M}'(\mathcal{C})$. Let us distinguish whether or not $p(\boldsymbol{c}) \in s_{next}$. If $p(\boldsymbol{c}) \notin s_{next}$ then $p(\boldsymbol{x}) \notin \mathsf{eff}'^+(op)$, as required. If $p(\boldsymbol{c}) \in s_{next}$ then $p(\boldsymbol{c}) \notin s$, which implies that that $p(\boldsymbol{x})$ is added to $\mathsf{eff}^+_!(op(\boldsymbol{c}))$ at line 20 and therefore $p(\boldsymbol{x}) \in \mathsf{eff}^+_{?!}(op)$, which contradicts the hypothesis that $p(\boldsymbol{x}) \notin \mathsf{eff}^+_{?!}(op)$. □

In the rest of the section, we study the properties of correctness and integrity for the learned models $\mathcal{M}$ and $\mathcal{M}^-_?$.

Theorem 2 (Correctness of $\mathcal{M}^-_?$)**.** *$\mathcal{M}^-_?$ correctly approximates $\mathcal{M}'$.*

Proof. Let s be a state of $\mathcal{M}^-_?(\mathcal{C}')$ for any set of constants $\mathcal{C}'$ possibly different from $\mathcal{C}$, and let $(s, op(\boldsymbol{c}), s^-_?) \in \delta^-_?$. By Lemmas 1-2, $\mathsf{pre}(op(\boldsymbol{c})) \supseteq \mathsf{pre}'(op(\boldsymbol{c}))$ and therefore there exists a tuple $(s, op(\boldsymbol{c}), s') \in \delta'$. We have that $s^-_? = s \cup \mathsf{eff}^+_!(op(\boldsymbol{c})) \setminus \mathsf{eff}^-_!(op) \setminus \mathsf{eff}^-_?(op)$. By Lemmas 1-4, $\mathsf{eff}^+_!(op(\boldsymbol{c})) \subseteq \mathsf{eff}'^+(op(\boldsymbol{c}))$; by Lemmas 1-5, $\mathsf{eff}'^-(op(\boldsymbol{c})) \subseteq \mathsf{eff}^-_{!?}(op(\boldsymbol{c})) = \mathsf{eff}^-_!(op(\boldsymbol{c})) \cup \mathsf{eff}^-_?(op(\boldsymbol{c}))$. Since $s' = s \cup \mathsf{eff}'^+(op(\boldsymbol{c})) \setminus \mathsf{eff}'^-(op(\boldsymbol{c}))$, it must be that $s^-_? \subseteq s'$. □

Theorem 3 (Correctness of $\mathcal{M}$)**.** *$\mathcal{M}(\mathcal{C})$ correctly approximates $\mathcal{M}'(\mathcal{C})$ from the final state of OLAM.*

Proof. Let s reachable from s_f in $\mathcal{M}(\mathcal{C})$. Suppose that $(s, op(\boldsymbol{c}), s'') \in \delta$. By Lemmas 1-2, $\mathsf{pre}(op(\boldsymbol{c})) \supseteq \mathsf{pre}'(op(\boldsymbol{c}))$ and therefore there exists a tuple $(s, op(\boldsymbol{c}), s') \in \delta'$. Suppose that $s'' \not\subseteq s'$, which implies that there is a $p(\boldsymbol{c}) \in s''$ which is not in s'. This can be caused by some missing negative effect in $\mathsf{eff}^-_!(op)$

or some extra positive effect in $\mathsf{eff}^+_!(op)$. The latter case is excluded by Lemma 4. Suppose that $p(\boldsymbol{x}) \in \mathsf{eff}'^-(op)$ but $p(\boldsymbol{x}) \notin \mathsf{eff}^-_!(op)$. From Lemma 5 we have that $\mathsf{eff}'^-(op) \subseteq \mathsf{eff}^-_{!?}(op)$. The fact that $\mathsf{eff}^-_!(op) \subseteq \mathsf{eff}'^-(op)$ implies $p(\boldsymbol{x}) \in \mathsf{eff}^-_?(op)$. Since s is reachable from the final state s_f via $\mathcal{M}$, then condition ((5.3)) must be false, which means that $\mathsf{eff}^-_?(op(\boldsymbol{c})) \cap s = \emptyset$, and therefore that $p(\boldsymbol{c}) \notin s$. Therefore if $p(\boldsymbol{c}) \in s''$, then it has been added by $p(\boldsymbol{x}) \in \mathsf{eff}^+_!(op) \subseteq \mathsf{eff}'^+(op)$. But this contradicts the fact that $p(\boldsymbol{x}) \in \mathsf{eff}'^-(op)$. □

Lemma 6. *At any execution step of OLAM if* $\phi \in \mathsf{pre}_\perp(op)$ *and* $\phi \neq \emptyset$ *then* $\phi \cap \mathsf{pre}'(op) \neq \emptyset$.

Proof. If $\phi \in \mathsf{pre}_\perp(op)$ and $\phi \neq \emptyset$, then ϕ has been added because of the failure of an action $op(\boldsymbol{c})$ in a state s and $\phi = \{p(\boldsymbol{x}) \in \mathsf{pre}(op) \mid p(\boldsymbol{c}) \notin s\}$. The failure of $op(\boldsymbol{c})$ in s implies that there is a $p(\boldsymbol{x}) \in \mathsf{pre}'(op)$ such that $p(\boldsymbol{c}) \notin s$. The fact that $\mathsf{pre}(op) \supseteq \mathsf{pre}'(op)$ implies that $p(\boldsymbol{x}) \in \phi$ and therefore $p(\boldsymbol{x}) \in \phi \cap \mathsf{pre}'(op)$. Hence we can conclude that $\phi \cap \mathsf{pre}'(op) \neq \emptyset$. □

Theorem 4 (Integrity of $\mathcal{M}$)**.** $\mathcal{M}(\mathcal{C})$ *integrally approximates* $\mathcal{M}'(\mathcal{C})$ *from the final state of OLAM.*

Proof. Suppose that s is reachable from the final state of OLAM via $\mathcal{M}(\mathcal{C})$ and that $op(\boldsymbol{c})$ is executable from s according to $\mathcal{M}'(\mathcal{C})$, i.e., that $\mathsf{pre}'(op(\boldsymbol{c})) \subseteq s$. First, let us show that $op(\boldsymbol{c})$ is also executable by $\mathcal{M}(\mathcal{C})$, i.e., that $\mathsf{pre}(op(\boldsymbol{c})) \subseteq s$. Suppose the contrary, i.e., that $\mathsf{pre}(op(\boldsymbol{c})) \setminus s \neq \emptyset$. Since s is reachable from s_f with $\mathcal{M}(\mathcal{C})$, the fact that OLAM terminates at s_f implies that condition ((5.1)) is false. This implies that $\mathsf{pre}(op(\boldsymbol{c})) \setminus s \in \mathsf{pre}_\perp(op(\boldsymbol{c}))$, which implies that $\phi = \{p(\boldsymbol{x}) \in \mathsf{pre}(op) \mid p(\boldsymbol{c}) \notin s\} \in \mathsf{pre}_\perp(op)$. Furthermore ϕ is not empty since $\mathsf{pre}(op(\boldsymbol{c})) \setminus s \neq \emptyset$. By Lemma 6 we have that there is $p(\boldsymbol{x}) \in \mathsf{pre}'(op)$ such that $p(\boldsymbol{c}) \notin s$, which implies that $op(\boldsymbol{c})$ is not executable in s by $\mathcal{M}'(\mathcal{C})$, which is a contradiction.

Let $(s, op(\boldsymbol{c}), s'') \in \delta$ and $(s, op(\boldsymbol{c}), s') \in \delta'$. Suppose by contradiction that $s' \not\subseteq s''$, which implies that there is a $p(\boldsymbol{c}) \in s'$ which is not in s''. This can be caused by some missing positive effect in $\mathsf{eff}^+_!(op)$ or some extra negative effect in $\mathsf{eff}^-_!(op)$. The latter case is excluded by Lemma 4. Suppose that $p(\boldsymbol{x}) \in \mathsf{eff}'^+(op)$ but $p(\boldsymbol{x}) \notin \mathsf{eff}^+_!(op)$. From Lemma 5 we have that $\mathsf{eff}'^+(op) \subseteq \mathsf{eff}^+_{!?}(op)$. The fact that $\mathsf{eff}^+_!(op) \subseteq \mathsf{eff}'^+(op)$ implies $p(\boldsymbol{x}) \in \mathsf{eff}^+_?(op)$. Since s is reachable from the final state via $\mathcal{M}$, condition ((5.2)) must be false and therefore $\mathsf{eff}^+_?(op(\boldsymbol{c})) \subseteq s$. This implies that $p(\boldsymbol{c}) \in s$. Therefore if $p(\boldsymbol{c}) \notin s''$, then it has been deleted by $p(\boldsymbol{x}) \in \mathsf{eff}^-_!(op) \subseteq \mathsf{eff}'^-(op)$. But this contradicts the fact that $p(\boldsymbol{x}) \in \mathsf{eff}'^+(op)$. □

The learned model $\mathcal{M}$ approximates the GTM from the final state s_f of OLAM *both* correctly and integrally. This implies that all and only the valid plans computed from s_f via $\mathcal{M}$ are valid plans from s_f via the GTM. Therefore, if a complete algorithm fails to reach a given set of goals from s_f via $\mathcal{M}$, then the goals cannot be reached also via the GTM.

5.5 Experimental Analysis

We evaluate the effectiveness of OLAM for online learning planning domains on 23 planning domains, including the domains from the learning tracks of the past IPCs and the domains used by [4]. For each domain, using an available problem generator, we randomly generated 10 small or middle-size instances with a number of objects ranging from 3 to 241 and consequently a number of potential grounded actions ranging from 12 to about $3.16 \cdot 10^6$. For every domain, OLAM is run on all the generated problem instances, from the smallest to the largest. On the first instance, OLAM takes as input the empty set of preconditions, positive and negative effects; for the successive runs, OLAM takes as input the planning domain $\mathcal{M}$ learned at the previous run. In OLAM, the calls EXECUTE and OBSERVE (lines 17-18) are implemented by a simulator of the IPC domain. The transition function of such a model is not known by the agent, who can only ask to execute actions and observe the current state. For function PLAN of Algorithm 2 (line 12), we adopt FASTDOWNWARD [50] with a 60 seconds timeout. All experiments were run on an Intel Xeon Skylake 2.3 GHz with 8 cores and 64 GB of RAM.

The learned planning domain is compared with the GTM, as done by [4], by precision and recall measures. Given a learned model $\mathcal{M}$ and GTM $\mathcal{M}'$, we define precision and recall for preconditions (P_{pre}, R_{pre}), positive and negative effects (P_{eff^-}, P_{eff^+}, R_{eff^-}, R_{eff^+}). Specifically, P_{pre} and R_{pre} are defined as follows:

$$P_{\mathsf{pre}} = \frac{\sum_{op} |\mathsf{pre}(op) \cap \mathsf{pre}'(op)|}{\sum_{op} |\mathsf{pre}(op)|} \quad R_{\mathsf{pre}} = \frac{\sum_{op} |\mathsf{pre}(op) \cap \mathsf{pre}'(op)|}{\sum_{op} |\mathsf{pre}'(op)|}.$$

Intuitively, they measure the (relative) amount of *extra* learned preconditions w.r.t. the GTM, and the (relative) amount of *missing* preconditions w.r.t. the GTM, respectively. The lower these amounts, the greater the measures. Similarly, we define precision and recall for eff^- and eff^+. If the precision and recall measures for pre, eff^- and eff^+ is 1, then the learned model is exactly the same as in the GTM for pre, eff^- and eff^+, respectively. The *overall* precision P and recall R are defined considering pre, eff^-, eff^+ together. I.e.,

$$P = \frac{\sum_{op} |\mathsf{pre}(op) \cap \mathsf{pre}(op)'| + |\mathsf{eff}^+(op) \cap \mathsf{eff}'^+(op)'| + |\mathsf{eff}^-(op) \cap \mathsf{eff}'^-(op)'|}{\sum_{op} |\mathsf{pre}(op)| + |\mathsf{eff}^+(op)| + |\mathsf{eff}^-(op)|},$$

and similarly for R.

5.5.1 Evaluation on IPC domains

Table 5.1 summarizes the efficacy of $\mathcal{M}$ w.r.t. the GTM in terms of precision and recall. By construction of sets $\mathsf{pre}(op)$ and $\mathsf{eff}_!^{+/-}(op)$ of every operator op, R_{pre}, P_{eff^+}, and P_{eff^-} of $\mathcal{M}$ must be equal to 1, i.e., there is no missing precondition and extra effect in the learned model $\mathcal{M}$ w.r.t. the GTM. This is confirmed by

Domain	$\#I$	P_{pre}	R_{pre}	P_{eff^+}	R_{eff^+}	P_{eff^-}	R_{eff^-}	P	R
barman	4	0.95	1	1	1	1	1	0.97	1
blocksworld	1	1	1	1	1	1	1	1	1
depots	1	0.94	1	1	1	1	1	0.97	1
driverlog	2	0.88	1	1	1	1	1	0.93	1
elevators	3	0.81	1	1	1	1	1	0.88	1
ferry	1	0.88	1	1	1	1	1	0.94	1
floortile	1	0.71	1	1	1	1	1	0.83	1
gold-miner	2	0.68	1	1	1	1	1	0.80	1
grid	2	0.71	1	1	1	1	1	0.82	1
gripper	1	1	1	1	1	1	1	1	1
hanoi	1	0.80	1	1	1	1	1	0.88	1
matching-bw	3	0.97	1	1	1	1	1	0.99	1
miconic	1	1	1	1	1	1	1	1	1
n-puzzle	1	0.75	1	1	1	1	1	0.88	1
nomystery	1	0.75	1	1	1	1	1	0.85	1
parking	2	0.78	1	1	1	1	1	0.89	1
rover	5	0.78	1	1	0.65	1	0.54	0.83	0.84
satellite	1	1	1	1	1	1	1	1	1
sokoban	1	0.80	1	1	1	1	1	0.89	1
spanner	1	0.90	1	1	1	1	1	0.94	1
tpp	3	0.94	1	1	1	1	1	0.97	1
transport	1	0.91	1	1	1	1	1	0.95	1
zenotravel	1	1	1	1	1	1	1	1	1

Table 5.1: Number of instances used to learn $\mathcal{M}$ (column 2), precision and recall over the preconditions, positive and negative effects of $\mathcal{M}$ (columns 3–8), overall precision and recall of $\mathcal{M}$ (columns 9-10).

the results in Table 5.1. Moreover, P_{pre} is always quite high, although usually lower than 1, i.e., there are few extra preconditions in the learned model w.r.t. the GTM. The extra-learned preconditions are static predicates such that, when the action is grounded, the corresponding grounded preconditions are true in all the states reachable from the initial state. This prevents the remotion of these extra preconditions from a correct learned model, like $\mathcal{M}$. The recall over the positive/negative effects is always equal to 1 for every domain but Rover, i.e., there are no missing effects (except for Rover) in the learned model w.r.t. the GTM.

The results in Table 5.1 also show that domain $\mathcal{M}$ can be learned using very few problems, often using only a single problem. Note that such a domain is learned by few small problems, and it does not mention their constants, i.e., it is general and hence suitable even for much larger problems. This shows that overall OLAM is able to effectively generalize between the experience derived from small environments and the future experience in large environments.

	with assumption				without assumption			
Domain	P_{eff^-}	R_{eff^-}	P	R	P_{eff^-}	R_{eff^-}	P	R
barman	1	1	0.97	1	0.24	1	0.56	1
blocksworld	1	1	1	1	0.43	1	0.69	1
depots	1	1	0.97	1	0.56	1	0.80	1
driverlog	1	1	0.93	1	0.23	1	0.53	1
elevators	1	1	0.88	1	0.15	1	0.42	1
ferry	1	1	0.94	1	0.50	1	0.75	1
floortile	1	1	0.83	1	0.10	1	0.29	1
gold-miner	1	0.82	0.80	0.95	0.18	1	0.41	1
grid	1	1	0.82	1	0.28	1	0.55	1
gripper	1	1	1	1	1	1	1	1
hanoi	1	1	0.88	1	1	1	0.88	1
matching-bw	1	1	0.99	1	0.32	1	0.65	1
miconic	1	1	1	1	0.23	1	0.62	1
n-puzzle	1	1	0.88	1	0.50	1	0.70	1
nomystery	1	1	0.85	1	0.10	1	0.30	1
parking	1	1	0.89	1	0.35	1	0.60	1
rover	1	0.54	0.83	0.84	0.16	0.54	0.55	0.84
satellite	1	1	1	1	0.67	1	0.92	1
sokoban	1	1	0.89	1	0.25	1	0.53	1
spanner	1	1	0.94	1	0.40	1	0.70	1
tpp	1	1	0.97	1	0.15	1	0.42	1
transport	1	1	0.95	1	0.33	1	0.65	1
zenotravel	1	1	1	1	0.33	1	0.67	1

Table 5.2: Precision and recall over the negative effects of $\mathcal{M}^-_?$ and overall model $\mathcal{M}^-_?$ with the assumption $\mathsf{eff}'^-(op) \subseteq \mathsf{pre}'(op)$ (columns 2–5), and without this assumption (columns 6–9).

We also study the efficacy of $\mathcal{M}^-_?$ w.r.t. the GTM. The difference between the learned models $\mathcal{M}$ and $\mathcal{M}^-_?$ consists in the fact that $\mathcal{M}^-_?$ also includes set $\mathsf{eff}^-_?(op)$ as negative effects of an operator op. Therefore, the precision and the recall over the preconditions and the positive effects of $\mathcal{M}^-_?$ are the same as in Table 5.1. Table 5.2 gives the precision and recall over the negative effects of $\mathcal{M}^-_?$ and over all domain $\mathcal{M}^-_?$. For this study we consider $\mathcal{M}^-_?$ with and without assuming $\mathsf{eff}'^-(op) \subseteq \mathsf{pre}'(op)$, i.e., when this assumption is made, the atoms in $\mathsf{eff}^-_?(op)$ that are not in the preconditions of an operator are removed. By construction of set $\mathsf{eff}^-_{!?}$, R_{eff^-} must be equal to 1. Surprisingly, this is false for domains Gold-miner and Rover. The reason why this happens is that for these domains an assumption of ours does not hold: Rover is a special domain including operators with inconsistent effects, i.e., $\mathsf{eff}'^+(op) \cap \mathsf{eff}'^-(op) \neq \emptyset$, for some operators. For Gold-miner, the assumption $\mathsf{eff}'^-(op) \subseteq \mathsf{pre}'(op)$ does not hold. This assumption is violated also in domains Parking, Satellite and

	OLAM			Fama			
Domain	Time	P	R	Time	P	R	Δ acts
blocksworld	5.03	**1**	**1**	510	**1**	**1**	-80
driverlog	20.42	**0.93**	**1**	349	0.79	0.85	-43
ferry	7.54	**0.94**	**1**	267	0.80	0.93	-85
floortile	47.34	**0.83**	**1**	517	0.82	0.78	-15
grid	36.92	**0.82**	**1**	306	0.81	0.74	-1
gripper	3.50	**1**	**1**	165	0.86	0.93	-89
hanoi	2.38	**0.88**	**1**	818	**0.88**	0.86	-96
miconic	4.24	**1**	**1**	200	0.81	**1**	-78
n-puzzle	1.97	**0.88**	**1**	23	0.86	**1**	-91
parking	183.94	**0.89**	**1**	895	0.84	0.84	-47
rover	154.10	**0.83**	**0.84**	629	0.51	0.53	175
satellite	11.26	**1**	**1**	65	0.70	0.89	-54
transport	74.98	**0.95**	**1**	280	0.80	0.89	-32

Table 5.3: CPU-seconds, precision and recall of OLAM (columns 2–4) and Fama (columns 5–7); difference between number of actions executed by Fama and OLAM (column 8): negative values mean that OLAM executes fewer actions. Bold values indicate best results.

MATCHING-BW, but for these domains there is no missing negative effect in $\mathcal{M}^{-}_{?}$, since OLAM on line 21 learns $\mathsf{eff}^{-}_{!}$ regardless of this assumption. Interestingly, P^{-}_{eff} with this assumption is always equal to 1, while without the assumption it is almost always quite low. This gap gives evidence that such an assumption can be very useful for removing extra negative effects from the learned domain.

We compare OLAM with a version of the algorithm that explores the world randomly. The random strategy reaches an average precision and recall of 0.45 and 0.63, against the average precision and recall of 0.99 and 0.92 obtained by OLAM, which shows that the generation of informative plan traces is extremely helpful.

5.5.2 Comparison with offline learning

In the last experiment we compare the online learning of OLAM with the offline learning method proposed by Fama [4]. Fama takes as input a set of plans with their state trajectories. Since OLAM does not support partial observability, we set Fama for working in a fully observable environment, and considered the same sets of plan traces and planning domains (but VISITALL and ZENOTRAVEL) as in [4]. The set of plan traces consists of 10 traces with 10 states; the set of planning domains does not contain ZENOTRAVEL and VISITALL, because the distributed version of Fama finds no solution for ZENOTRAVEL, and there is no problem generator available for VISITALL. Since Fama adopts the assumption $\mathsf{eff}'^{-}(op) \subseteq \mathsf{pre}'(op)$ for any operator op, we compared the planning domain derived from Fama with $\mathcal{M}^{-}_{?}$ using the same assumption. We obtained similar

results from the comparison between Fama and the other learned domain $\mathcal{M}$.

Table 5.3 compares OLAM and Fama. OLAM obtains better or equal precision and recall, and generally it is also much faster. In all the domains but Rover, OLAM executes less actions than Fama. We think that the difference for Rover is related to the consistent-effects assumption made in OLAM that in Rover does not hold. Overall, learning the planning domain online is much more effective than learning it offline. In our online approach, indeed, the agent selects the goals to reach and actions to execute to optimize learning, while in offline approaches actions are provided in the input traces.

Chapter 6

Online Grounding of Action Models

Symbolic planners are powerful and flexible tools that, given a general symbolic description of an available set of actions (i.e., a planning domain) and a detailed description of an environment, are capable of generating plans for achieving ideally any goal about (known) objects in the environment. In several applications, the information about the environment required to instantiate a planning domain is not available from the beginning. In particular, when an agent is placed in a new environment, it does not know the objects that populate the environment, and therefore it does not know their specific properties and relations. Consider, for instance, a robot that has to move around and manipulate objects in a kitchen (tables, chairs, apples, etc.) without knowing which and how many objects are really in the room. In this setting, the exploitation of a planning domain is a compelling challenge for three main reasons. First, in realistic environments, it is unfeasible for the robot to acquire a complete/correct and sufficiently detailed description of the environment before starting to plan and execute actions towards the achievement of its goals. Second, a robotic agent usually has a first-person perspective and partial view of the environment (e.g., by an on-board camera), hence the only way to acquire symbolic knowledge suitable for planning is by executing actions, observing their effects through its sensors, and mapping the sensory data (e.g., raw images) in a symbolic state. Third, high-level actions of the planning domain are not directly executable by the robot, and therefore they need to be compiled to low-level actions executable by the robot's actuators. For instance, given an object instance identified by a constant c_0, the action $\mathsf{goCloseTo}(c_0)$ is compiled into a sequence of robot movements and rotations, which follows the path provided by a path-planner, and moves the robot to the (nearest) location close to object c_0.

In the agent-environment interface (Figure 6.1), OGAMUS focuses on the perception function, since, though not being learned, an incorrect perception function is used for mapping the agent's perception into a symbolic state suitable

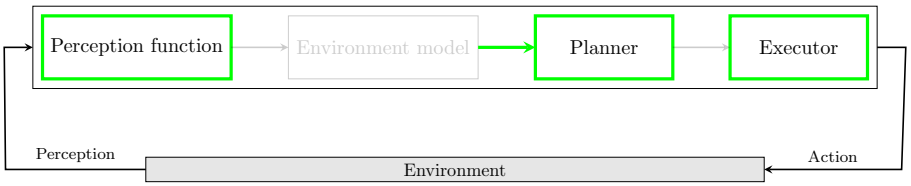


Figure 6.1: OGAMUS within the agent environment interface.

for applying symbolic planning. The planner module is used for computing a symbolic plan that leads the agent toward the achievement of a given goal, and the executor module is considered since the agent has to execute the symbolic actions in the plan through its actuators. In OGAMUS, less importance is given to the environment model, since a symbolic planning domain is required as input.

6.1 The Ogamus Framework

We propose a framework for agents that incrementally instantiate a planning domain, specified in PDDL, by planning, acting, and sensing, in an unknown environment. At each time point, the belief of the agent about the current state of the environment is represented by three components, namely: (i) the set of objects currently known by the agent and their properties expressed with the predicates of the PDDL domain, e.g., $\mathsf{table}(c_0)$, $\mathsf{apple}(c_1)$, and $\mathsf{on}(c_1, c_0)$; (ii) for each known object, a set of low-level features as perceived by the agent, e.g., visual features and positions of c_0 and c_1; (iii) a set of global features associated with the current environment state, e.g., an occupancy map of the environment, and the current pose of the robot.

For this framework, we propose an online iterative algorithm, called OGAMUS (Online Grounding of Action Models in Unknown Situations), which allows an agent, equipped with a lifted PDDL planning domain, and placed in an unknown environment, to achieve a set of goals expressed in the language of its PDDL action model. The agent is initialized without prior knowledge about the environment where it has to operate, i.e., with the empty set of objects, the empty set of their properties, and all the points of the occupancy map set as traversable. OGAMUS attempts to achieve the goal by combining four main activities, namely: (i) *exploring the environment* to acquire the knowledge needed to achieve the goals; (ii) *abstracting the sensor information* obtained at every step into a symbolic state; (iii) performing *symbolic planning* in the abstract model grounded with the current beliefs and current abstract state; (iv) *executing* the planned abstract actions by compiling them into low-level operations suitable for the current state of the environment.

The main features of OGAMUS are the following. *Generality:* OGAMUS is able to deal with any goal that can be expressed by a (first-order) formula using the predicates of the PDDL domain. For instance the goal of putting "two

apples on a table" can be specified by the formula $\exists x\, y\, z.\mathsf{on}(x, z) \wedge \mathsf{on}(y, z) \wedge \mathsf{apple}(x) \wedge \mathsf{apple}(y) \wedge \mathsf{table}(z) \wedge x \neq y$. Notice that goals are expressed with existentially quantified variables; this is because, initially, the agent is not aware of any object in the domain. An important step, necessary to achieve a goal containing existential variables, concerns discovering the object instances of the proper types (apple and table in the previous example) for instantiating all the existential variables. *Explainability*: The behaviour of the agent, its plans, and the effects of actions are represented at a symbolic level in which the states of the PDDL domain are derived at every step by abstracting the sensory data. *Robustness:* The action model, the obtained symbolic state representation, and the action compilation are not required to be fault free. As experimentally shown in this paper, OGAMUS achieves a high success rate even with low-precision object detectors and classifiers.

In order to use an abstract model, an agent needs to *anchor* the symbols occurring in the states of the planning domain with real-world perceptions, and to map abstract actions into actions executable in the real world [21]. We suppose that the agent can *partially* observe the current state of the environment through a set of sensors, for instance images provided by an RGB-D camera, which do not directly correspond to the states of the abstract model. Furthermore, the set of sensors provides only a partial and subjective view of the environment. For instance, the RGB-D camera provides only an egocentric view of a portion of the room visible by the agent. We also suppose that the agent interacts with the environment by executing low-level operations (e.g., move 25 cm forward, rotate 30° left, pick up or put down an object at the GPS-coordinates (x, y, z)), which are different from the actions in the abstract action model. We need therefore to link the abstract state to real perceptions, and the abstract actions to operations executable by the actuators of the agent. Let us first consider the relationship between abstract states and perceptions.

Object and state anchoring. Every object that the agent is aware of at a given instant is represented by a constant $c \in \mathcal{C}$ that is the internal identifier for such an object. Following the approaches to symbol anchoring proposed in the literature [21, 88], every constant $c \in \mathcal{C}$ is associated with a tuple of numeric features denoted by $\boldsymbol{z}_c$. For instance, $\boldsymbol{z}_c$ might include the estimated position of c and a set of visual features of the different views of c. In addition, for each state s determined by the agent, we have a vector of state features $\boldsymbol{z}_s$, consisting of the 3D position of the agent in the environment, the orientation of the agent relative to its initial pose, the information about the success of the last low-level operation made by the agent, and an occupancy map of the environment. The occupancy map is a 2D map of the environment storing the areas that are believed to be traversable by the robot. The occupancy map is initialized so that every point is traversable.

Predicate predictors. In order to map the perceptions about objects into atoms of the symbolic state, the agent associates to every predicate a

probabilistic model, e.g., a neural network, that computes the probability of a certain atom $P(\boldsymbol{c})$ to be true given the features associated to $\boldsymbol{c}$ and the current state ones, i.e., $Pr(Y_{P(\boldsymbol{c})} = \text{True} \mid \boldsymbol{z_c}, \boldsymbol{z}_s)$, where $Y_{P(\boldsymbol{c})}$ is a boolean random variable associated to the atom $P(\boldsymbol{c})$. These probabilistic models can be updated during execution on the basis of new observations. In this paper, however, we suppose that these probabilistic models are given (e.g., a pre-trained neural network), and they are not modified during execution.

We call *belief state* the agent's knowledge about object/state anchoring and predicate predictors.

Definition 14. *An agent belief state is a 5-tuple* $\langle \mathcal{C}, \boldsymbol{z}_\mathcal{C}, s, \boldsymbol{z}_s, \mathbf{Pr} \rangle$ *where:*

- $\mathcal{C}$ *is a set of constants;*
- $\boldsymbol{z}_\mathcal{C} = \{\boldsymbol{z}_c\}_{c \in \mathcal{C}}$ *is a set of object feature vectors* $\boldsymbol{z}_c$*;*
- $s \subseteq \mathcal{P}(\mathcal{C})$ *is the set of atoms that are believed to be true;*
- $\boldsymbol{z}_s$ *is a vector of state features;*
- $\mathbf{Pr} = \{Pr(Y_{P(\boldsymbol{c})} \mid \boldsymbol{z_c}, \boldsymbol{z}_s)\}_{P \in \mathcal{P}}$ *is the set of probabilistic models used to predict the truth value of* $P(\boldsymbol{c})$ *given the features* $\boldsymbol{z}_s$ *and* $\boldsymbol{z_c}$ *associated with the constants in* $\boldsymbol{c}$*.*

6.2 The Ogamus Algorithm

So far, we have not considered how the set $\mathcal{C}$ of constants identifying objects is obtained by the agent. We do not assume that they are given a priori to the agent; instead, we are interested in providing the agent with the capability to discover objects by adding new constants to the representation of the environment, updating the anchor to an object, merging two constants anchored to the same object, and deleting a constant from the representation that was erroneously identifying a non-existing object in the environment.

Let $\mathbf{x}$ be the vector that contains the data returned by the sensors (i.e., the observations) at a given time; the agent extracts from $\mathbf{x}$ a set of objects $\mathcal{C}_\mathbf{x}$, and for each object $c \in \mathcal{C}_\mathbf{x}$ a feature vector $\boldsymbol{z}_c$. Since the agent can also recognize objects that it has already seen, it is possible that $\mathcal{C}_\mathbf{x} \cap \mathcal{C} \neq \emptyset$.

Algorithm 3 OGAMUS algorithm

Input: $\mathcal{M}$, $\mathcal{G}$, **Pr** and MAXITER $\in \mathbb{N}$.
Output: SUCCESS/FAIL

1: $\langle \mathcal{C}, \boldsymbol{z}_{\mathcal{C}}, s, \boldsymbol{z}_s \rangle \leftarrow \langle \emptyset, \emptyset, \emptyset, (\mathbf{0}, nil, \emptyset) \rangle$
2: **for** $1 = 0$ **to** MAXITER **do**
3: **if** $s \models \mathcal{G}$ **then**
4: **return** SUCCESS
5: **end if**
6: $\pi \leftarrow \text{PLAN}(\mathcal{M}, \mathcal{C}, s, \mathcal{G})$
7: **if** $\pi = \text{NONE}$ **then**
8: $\boldsymbol{e} \leftarrow \text{EXPLORE}(\boldsymbol{z}_s)$
9: **else**
10: $op(\boldsymbol{c}) \leftarrow \text{POP}(\pi)$
11: $\boldsymbol{e} \leftarrow \text{COMPILE}(op(\boldsymbol{c}), \boldsymbol{z}_{\boldsymbol{c}}, \boldsymbol{z}_s)$
12: **end if**
13: $e_1 \leftarrow Pop(\boldsymbol{e})$
14: $\mathbf{x} \leftarrow \text{EXEC}(e_1)$
15: $\boldsymbol{z}_s \leftarrow \text{GETSTATEFEATURES}(\mathbf{x})$
16: $\mathcal{C}_{\mathbf{x}}, \boldsymbol{z}_{\mathcal{C}_{\mathbf{x}}} \leftarrow \text{GETOBJS}(\mathbf{x})$
17: $\mathcal{C}, \boldsymbol{z}_{\mathcal{C}} \leftarrow \text{UPDATEOBJS}(\mathcal{C}, \boldsymbol{z}_{\mathcal{C}}, \mathcal{C}_{\mathbf{x}}, \boldsymbol{z}_{\mathcal{C}_{\mathbf{x}}})$
18: $Pr(\boldsymbol{Y}_{\mathcal{P}(\mathcal{C})}) \leftarrow \text{PREDICTSTATE}(\boldsymbol{z}_{\mathcal{C}}, s, \boldsymbol{z}_s)$
19: $s \leftarrow \{p(\boldsymbol{c}) \in \mathcal{P}(\mathcal{C}) \mid Pr(Y_{p(\boldsymbol{c})} = True \mid \boldsymbol{z}_{\boldsymbol{c}}) > 1 - \epsilon\}$
20: **if** $\pi \neq$ NONE and SUCCEED$(op(\boldsymbol{c}))$ **then**
21: $s \leftarrow s \cup \mathsf{eff}^+(op(\boldsymbol{c})) \setminus \mathsf{eff}^-(op(\boldsymbol{c})))$
22: **end if**
23: **end for**
24: **return** FAIL

In the following, we shortly describe the OGAMUS algorithm (Algorithm 3).

- The algorithm takes as input an action model $\mathcal{M}$, a set **Pr** of probabilistic models for predicting the predicates in $\mathcal{P}$, a goal formula $\mathcal{G}$, and a maximum number of iterations. Notice that the goal $\mathcal{G}$ cannot contain constants, since we suppose that at the beginning the agent is not aware of any object. For instance, the goal requiring that an apple is inside a box can be encoded by the PDDL expression representing formula $\exists x, y\ \mathsf{apple}(x) \wedge \mathsf{box}(y) \wedge \mathsf{in}(x, y)$.
- The agent starts by initializing all the components of its state to the empty set (line 1). We assume indeed that the agent is not aware of any object in the environment, therefore $\mathcal{C} = \emptyset$. Since $\mathcal{C}$ is empty, $\boldsymbol{z}_C$, $\mathcal{P}(\mathcal{C})$ and s are also empty. The information in $\boldsymbol{z}_s$ representing the position and orientation of the agent is initialized with a vector of 0's; the information in $\boldsymbol{z}_s$ about the success of the last operation is set to *nil*; finally, the occupancy map of the environment in $\boldsymbol{z}_s$ is set to an empty map so that all the points are traversable.

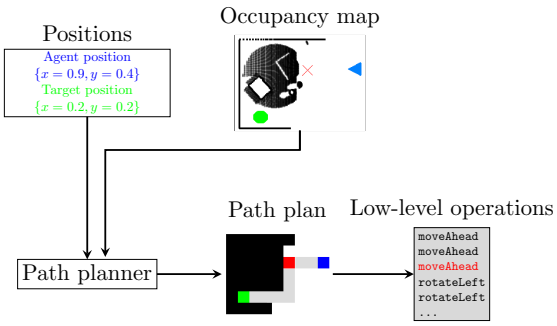


Figure 6.2: Example of EXPLORE($\boldsymbol{z}_s$). The occupancy map, agent position (in blue), and target position (in green) are given as input to a path planner which discretizes the occupancy map, computes a path plan, and compiles the path plan into a sequence of low-level operations.

– Then the agent iterates for a maximum number of steps, checking if the current state s satisfies the goal (line 4); when this is the case, it returns SUCCESS.

– Otherwise, the agent invokes a planner (line 6) to solve the planning problem defined on the input action model, the current set of objects, the current state, and the input goal formula $\mathcal{G}$.

– If the planner does not find a plan that satisfies the goal, then the agent explores the environment in order to discover new objects that are needed to satisfy the goal. For instance, if the goal is to put an apple into a box, then the planner can find a plan only if in the current state s there is at least one object of type apple and one of type box. For the exploration phase (line 8), the agent randomly selects a target position on the occupancy map (stored in $\boldsymbol{z}_s$) that it believes to be free from other obstacles. As shown in Figure 6.2, EXPLORE($\boldsymbol{z}_s$) calls a path planner that checks if such a position is reachable (if it is not reachable a new position is selected) and returns a sequence $\boldsymbol{e}$ of low-level navigation and rotation operations, which, according to the current knowledge of the agent, moves the agent from its current position to the selected target. For efficiency reasons, this path is computed in an approximated occupancy map obtained by discretizing the occupancy map through a grid. The execution of such a sequence of operations might fail due to the partial or incorrect knowledge of the agent, i.e., when the agent wrongly believes that a certain area on the path returned by the path-planner is traversable while there is an obstacle. In Figure 6.2, the agent fails to reach the red cell. Indeed, the execution of the third moveAhead action fails because the robot bumps into a table, whose size was only partially determined at the beginning of the exploration phase. The exploration terminates whenever the goal is reachable according to the learned problem. For example, consider the

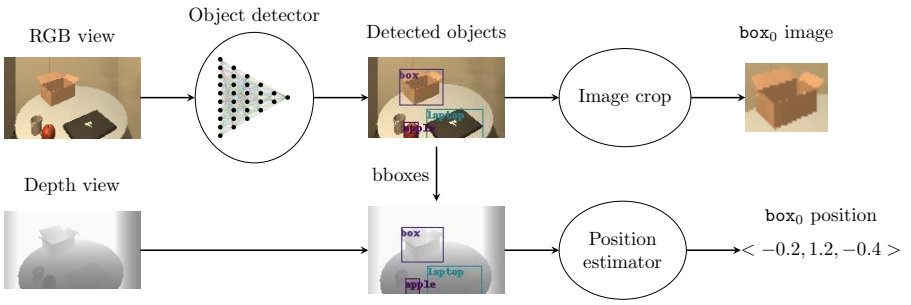


Figure 6.3: Example of object features extraction. The object detector takes as input the perception composed by the agent view RGB image, and returns a set of bounding boxes together with the detected object types. The object bounding boxes are used with the agent view depth image to estimate their positions (in meters w.r.t. the initial agent position). The lighter the pixels in the box, the farther the associated object.

goal requiring that an apple is in a box and assume that an apple has already been discovered in the environment. The execution of the computed sequence of operations terminates in advance, when the agent detects the presence of a box on the table in Figure 6.2, as it approaches the table by executing the first operations in the sequence.

– If instead the planner succeeds and returns a valid plan π, then the first action of π is compiled into a sequence of low-level operations $\boldsymbol{e}$ (line 11). The compilation of the action is based on the object and state features available in the agent's state. For instance, the high-level action $\mathsf{pickup}(c)$ is compiled into the low-level operation $\mathsf{pickup}(x, y, z)$ where (x, y, z) is the current (believed) position of object c, memorized in $\boldsymbol{z}_c$. The simulator executes such an operation by picking up what is present at these coordinates. To compile the action $\mathsf{goCloseTo}(c)$ instead, the agent calls a path planner that provides a path from the current position of the agent (memorized in $\boldsymbol{z}_s$) to a position close to c.

– Successively, the first operation of sequence $\boldsymbol{e}$ is executed (line 14), and a new observation $\boldsymbol{x}$ is obtained. The execution of the first operation may fail or not. In both cases, the agent can acquire new knowledge (e.g., discover new objects or an obstacle), which can be used to produce a better compilation of a high-level action, and/or produce a better plan. Then, the new state features $\boldsymbol{z}_s$ are extracted from the sensory data $\mathbf{x}$ (line 15). The information about the occupancy map is updated using the information of success/failure of the action and the depth image.

– Then the agent runs an object detector (line 16) on the RGB image contained in the observation $\mathbf{x}$, which returns a set of objects $\mathcal{C}_{\mathbf{x}}$, each associated with a vector of numeric features $\boldsymbol{z}_c$. These features include the

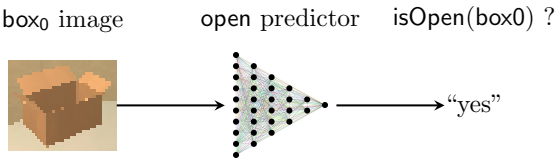


Figure 6.4: Example of predictor for the open predicate. The predictor takes as input the RGB images associated to box_0, and returns the predicted truth value of $\mathsf{isOpen}(\mathsf{box}_0)$.

bounding box, an estimation of the object position, and a vector of visual features extracted from the cropping of the image with the bounding box. Figure 6.3 gives an example of extraction of a number of objects, including a box, from the egocentric view of an agent robot, together with the bounding-box image of such a box and the estimate of its position.

– Next, at line 17, the agent merges the objects $\mathcal{C}_\mathbf{x}$ recognized in the current perception with the ones already known, i.e., $\mathcal{C}$. For every object $c' \in \mathcal{C}_\mathbf{x}$ there are two possible situations: (i) c' does not match with any object $c \in \mathcal{C}$, and therefore it is added to $\mathcal{C}$ with the corresponding features $\boldsymbol{z}_{c'}$; (ii) c' matches with a $c \in \mathcal{C}$; in this case the features $\boldsymbol{z}_c$ of c are extended/updated with the features $\boldsymbol{z}_{c'}$. In the implementation, we use a very simple matching criterion which considers only the estimated position of the objects. Two objects are matched when their distance is less than a given threshold (set to 20cm). More sophisticated criteria can be adopted by defining a suitable distance measure between the entire set of object features. However, this simple criterion turned out to be sufficiently effective in our experiments.

– In line 18, the agent predicts the truth values of each atom in $\mathcal{P}(\mathcal{C})$ for the updated set of constants $\mathcal{C}$ by applying the predictors $\mathbf{Pr}$ on the features $\boldsymbol{z}_\mathcal{C}$. For predicate $\mathsf{closeToAgent}$, the prediction takes also as input the agent position in $\boldsymbol{z}_s$. All the atoms involving new or merged objects must be evaluated; the remaining atoms are evaluated only if the corresponding predictor takes as input some feature that has been updated after the execution of the last action. For instance, if the agent executes a move action, then all the atoms $\mathsf{closeToAgent}(c)$ for all $c \in \mathcal{C}$ must be evaluated. Each atom $\mathsf{closeToAgent}(c)$ is predicted true if the euclidean distance between the position of the object represented by c in z_c and the agent position in z_s is lower than a given threshold (set to 140 cm). When the action $\mathsf{open}(\mathsf{box}_0)$ is executed, the visual features of box_0 probably change, and the truth value of predicate $\mathsf{isOpen}(\mathsf{box}_0)$ is predicted as depicted in Figure 6.4. Notice that, after executing an open operation it is not guaranteed that the object will be open, as the action might fail.

– At line 19, the new state s is created with all predicates $P(\boldsymbol{c})$ such that

$Pr(Y_{P(\boldsymbol{c})} = True \mid \boldsymbol{z_c}, \boldsymbol{z_s})$ is higher than a given threshold $1 - \epsilon$ with $\epsilon \in [0, 1]$. Our approach does not assume to have access to the correct abstract state. Indeed, the agent can produce inconsistent states (e.g., a box is both on the table and on another box), or states that do not comply with action effects. Inconsistent states do not prevent OGAMUS to further planning and, whether a failure occurs, revise the agent's knowledge making them consistent. In the second case, the agent monitors the execution of high-level actions by comparing the state predicted by the PDDL model with the perceived state, and it solves inconsistencies in favor of the action effects in the model.

- At line 21, when SUCCEED($op(\boldsymbol{c})$) is true, i.e., the entire sequence $\boldsymbol{e}$ of operations compiling the first action $op(\boldsymbol{c})$ of π is successfully executed, the state s is updated according to the effects of $op(\boldsymbol{c})$.
- If the agent does not reach a state s that satisfies the goal $\mathcal{G}$ after MAXITER steps, then the algorithm returns FAIL (line 24).

Knowledge Revision for New Tasks To show the generality and the modularity of the proposed framework, we describe how it can be easily extended to cope with new tasks that can possibly involve a set of new (PDDL) actions, predicates, and object types. To allow the agent to accomplish a new task t_{new}, we firstly need to encode t_{new} in a PDDL goal formula. If the encoding of t_{new} does not require the introduction of new predicates, actions, or object types, then to solve the task it is sufficient to invoke OGAMUS with the goal formula encoding t_{new}. If, instead, the encoding of t_{new} requires some new predicates, actions, or object types, then we have to provide the agent with the capability of (1) recognizing objects of the new types, (2) predicting the truth value of the new predicates, and (3) compiling the new actions in low-level operations executable by the agent's actuators. Consider, for instance, the case where t_{new} is the task turning a lamp on: t_{new} can be specified by the goal formula $\exists x.\mathsf{lamp}(x) \wedge \mathsf{turned\_on}(x)$, where lamp is a new object type and turned_on a new unary predicate.

To detect objects with the newly introduced type e.g., lamp, the object detector need to be extended and retrained with the new object type. This implies that the upper part of the detector (which is responsible for classifying the objects in their types) need to be extended with the new type and re-trained on a dataset containing also examples for this new type. The introduction of a new predicate, in our example turned_on, requires the deployment of a new classifier that predicts if the predicate holds for the objects detected from the sensory data. In case the predictor is based on a supervised learning model, then a training dataset with objects labeled with positive and negative examples of the predicate need to be provided. Finally, if the new task requires adding new actions to the PDDL model, (e.g., to make the predicate $\mathsf{turned\_on}(x)$ true/false we need to introduce two new action turn_on and turn_off) we need to specify how the new action can be compiled into a sequence of low-level operations

executable by the agent. For example, action $\mathsf{turn\_on}(c)$ will be compiled into a low-level operation $\mathsf{turn\_on}(x, y, z)$ where (x, y, z) is the believed position of the object c.

6.3 Experimental Analysis

We perform two sets of experiments. First, we experimentally evaluate OGAMUS with a simulated environment on four tasks that involves going close and moving objects present in a number of rooms. Then, we compare OGAMUS with a state-of-the-art approach on the specific task of object goal navigation in different apartments.

6.3.1 Evaluating Ogamus

The tasks and the corresponding goals on which we evaluate OGAMUS are the following:

1. Object goal navigation (OBJNAV t_1): given an object type t_1, the agent has to find, go close to, and look at an object of type t_1. For instance, the agent has to go close to an apple and look at it. The corresponding goal is $\exists x(\mathsf{Apple}(x) \wedge \mathsf{CloseToAgent}(x) \wedge \mathsf{Visible}(x))$. The agent is close to an object when the distance from the object is less than 1.5 meters.

2. Open/close an object (OPEN/CLOSE t_1): the agent is required to go close to an object of type t_1, look at it, and open/close it. For instance, the agent has to open a drawer; the corresponding goal is $\exists x(\mathsf{Drawer}(x) \wedge \mathsf{Open}(x))$. In order to manipulate an object the agent need to be at a distance less than 1.5 meters.

3. Stack an object of type t_1 on an object of type t_2 (ON t_1 t_2): the agent has to find two objects of types t_1 and t_2 and put the one of type t_1 on top of the other of type t_2. For instance, the agent has to put an apple on a table. The corresponding goal is: $\exists xy(\mathsf{Apple}(x) \wedge \mathsf{Table}(y) \wedge \mathsf{On}(x, y))$.

Since these tasks do not involve numerical resources or temporal constraints, we adopted propositional PDDL planning.

Simulator. We used the ITHOR [64] simulator, an open-source interactive environment for Embodied AI. ITHOR provides 120 different scenes, such as kitchens, living rooms, bathrooms, and bedrooms, and allows a realistic simulation of the environment, including the physics of the objects. The scenes contain objects of 118 different types. The agent perceives the current state of the environment through an RGB-D on-board camera that provides a photo-realistic rendering of its egocentric view. The agent also perceives its position and orientation via a GPS and a compass (relative to the initial pose, which is unknown). The agent can navigate the environment by moving ahead of a

given distance (set to 25cm), turning left or right, and looking up or down by a given angle (set to 30°).[1] The agent can pick up objects, move them around, and change their state (e.g., a fridge can be opened or a laptop switched on).

For the object goal navigation task, we also considered a second simulator, RoboThor [27]. RoboThor is another simulation environment designed to develop embodied AI agents. Recently, RoboThor hosted a competition that tackles an object goal navigation challenge; in our experiments, we also compared Ogamus with the approaches that took part in the competition.

Object detector. As an object detector, we used the Faster-RCNN model available in PyTorch 1.9 [87], pre-trained on the COCO dataset [76] and fine-tuned on a self-generated dataset. In addition to the bounding box of the detected object, the object detector returns also the classification in one of the 118 classes. The object detector has been trained on a dataset composed by 69,095 training and validation images. The labeling of the dataset has been done by using the ground truth provided by iThor. We tested it on 12,892 images obtaining a precision and recall of 50.99% and 65.18%, respectively.

Predicate predictors. For predicting predicate On, we trained a feed-forward neural network [111] with 244 input features composed by the bounding boxes coordinates of the two objects involved in the predicate relation and the 1-hot encoding of the two predicted classes returned by the object detector. For such a predicate, the training (and validation) sets is composed of 36,344 labeled pairs of objects. We evaluate the prediction of predicate On on a test set composed of 8678 object pairs, obtaining 98.32% of both precision and recall. For predicting the unary predicate Open, we used a ResNet50 neural network [49] to extract features from the cropped object image, followed by a linear layer with input size 2048.[2] We trained it on 48,476 labeled examples, and test it on 9685 examples, obtaining 92.84% precision and 92.54% recall. The unary predicate closeToAgent, meaning that the agent is near to the object mentioned by the predicate, is computed directly from the features of the object. Specifically, we check if the distance between the agent position memorized in $\boldsymbol{z}_s$ and the object position memorized in the object feature vector is less than the manipulation distance, which is set to 1.5 meter in iThor and 1 meter in RoboThor. Finally, we have to predict the equality predicate, i.e., when two objects c and d with features $\boldsymbol{z}_c$ and $\boldsymbol{z}_d$ represent the same object. For this purpose, we compute the distance between the two estimated object positions, and assign the object features to the same object instance whether such a distance is lower than a given threshold (set to 20 cm in our experiments). All the training, validation, and testing data have been extracted from a set of images collected by navigating in the iThor simulator.

[1] These settings are those indicated by the simulator developers for their proposed challenges.

[2] Further technical details about the hyper-parameters and datasets are available in the supplementary material.

	Success ↑		*DTS* ↓		$P_{\mathcal{C}}$ ↑		$R_{\mathcal{C}}$ ↑		$P_{\mathcal{P}}$ ↑		$R_{\mathcal{P}}$ ↑	
	$\mathcal{C}$	$\mathcal{C}_{GT}$	$\mathcal{C}$	$\mathcal{C}_{GT}$	$\mathcal{C}$	$\mathcal{C}_{GT}$	$\mathcal{C}$	$\mathcal{C}_{GT}$	$\mathcal{C}$	$\mathcal{C}_{GT}$	$\mathcal{C}$	$\mathcal{C}_{GT}$
On	0.5	0.8	1	0.37	0.28	1	0.86	1	0.83	0.82	0.8	0.87
Open	0.75	0.87	0.45	0.25	0.35	1	0.78	1	0.82	0.81	0.72	0.82
Close	0.78	0.89	0.39	0.16	0.32	1	0.8	1	0.8	0.79	0.73	0.82
objNav	0.78	0.83	0.27	0.19	0.42	1	0.8	1	0.82	0.8	0.75	0.84

Table 6.1: Performance of Ogamus with/out the ground-truth object detection, evaluated on the considered tasks in the iThor simulator. ↑/↓ means the higher/lower the better.

Evaluation metrics. The evaluation is provided by calculating a number of standard metrics over a set of episodes. For each task, an episode is obtained by randomly placing the agent in a random unseen scene and providing it with a randomly generated goal for the given task. The generated goals are feasible since the object types used in their definitions are randomly chosen from a proper set of types; e.g., the goal to open a box is defined by randomly choosing the type "box" from the set of object types that can be opened. For all the tasks we adopt the following standard evaluation metrics:

Success rate (Success): is equal to the fraction of successful episodes on the total number of episodes.

Distance To Success (DTS): For tasks (ObjNav t_1), (Open t_1), and (Close t_1), it is the average distance between the agent and the closest object of type t_1; for the task (On $t_1\, t_2$), it is the average distance between the closest pair of objects of types t_1 and t_2. If the episode succeeds such a distance is set to 0.

In order to measure the impact of errors in object detections, for each task we consider two versions of Ogamus. In the first version, the set of objects $\mathcal{C}$ is those returned by our object detector; in the second version, the set of objects $\mathcal{C}_{GT}$ is those returned by the iThor simulator, which corresponds to a ground truth object detector. Moreover, for all tasks, we evaluate the precision $P_{\mathcal{C}}$ and recall $R_{\mathcal{C}}$ of the detected objects, and the precision $P_{\mathcal{P}}$ and recall $R_{\mathcal{P}}$ of their predicate relations. $P_{\mathcal{P}}$ and $R_{\mathcal{P}}$ take into account only the objects that match with ground-truth ones. The matching is performed by computing the Intersection over Union (IoU) among the 2D bounding box detected during the episode and the ground-truth ones: if the IoU is higher than 50% for a ground-truth object of the same class, then the detected object matches with it.

Experimental results. In our experiments, a run of Ogamus consists of 200 steps, where at each step a low-level operation is performed; we call each of these runs an episode. For all tasks, the episode dataset uses the *test* scenes of iThor, i.e., all environments that does not appear in the datasets generated for training the predicate classifiers and object detector.

In Table 6.1, we report the average results of all tasks with and without ground-truth object detection over the considered episodes. For task ON, we randomly generated 400 different goals, defining 400 episodes; for tasks OPEN and CLOSE, we randomly generated 100 goals, defining 100 episodes for each task; for the object goal navigation task, we used the test set of goals proposed in [119], defining 2133 episodes. It is worth noting that, for the object goal navigation task, two different episodes often have the same goal but a different initial pose of the agent.

The impact of errors in object detecting for tasks OBJNAV, OPEN and CLOSE is pretty low and, as expected, it is half of the impact for task ON, since this latter task requires to detect two objects, while all other tasks require to detect a single object. Without ground-truth object detection, OGAMUS achieves the best success rate on the object goal navigation task; same or similar results are also provided in tasks OPEN and CLOSE, since they can be seen as an extension of the object goal navigation task where, after finding and going near to an object, the agent has only to open or close the object. In the ON task, the success rate decreases significantly, because it requires moving towards two objects, instead of only one, and has two additional complexities given by the facts that one object must be placed on the other one in a clear place, i.e., a place not obstructed by other objects, and that the total encumbrance of the agent increases when it carries an object, which causes more collisions during the navigation.

Metric $P_{\mathcal{C}}$ measures the amount of false positive object detections. Although the value of $P_{\mathcal{C}}$ is quite low for almost all the tasks, the success rate is relatively high because: (i) many false positive objects are not involved in the definition of the goals; (ii) the agent acts by using the objects with the highest confidence, which usually correspond to ground truth objects. $P_{\mathcal{C}}$ is higher for the object goal navigation task, because in this task the agent achieves the goal in fewer steps than for other tasks, and this reduces the number of predictions and the chance of detecting false positive objects.

Metric $R_{\mathcal{C}}$ measures the amount of true positive detected objects. The values for $R_{\mathcal{C}}$ are quite high, and hence the real existing objects are often detected, although in our experiments the agent sometimes fails to recognize objects when they are far from the agent. Moreover, the values of $P_{\mathcal{P}}$ and $R_{\mathcal{P}}$ are relatively high, and hence the agent can construct a symbolic state that is quite correct and complete, enabling effective planning.

As expected, when OGAMUS is provided with ground-truth object detection, all metrics are better than or similar to using our object detection. Only $P_{\mathcal{P}}$ is slightly lower when ground-truth object detection is used; we think this is due to the fact that sometimes the ground-truth object detection identifies objects which are only partially seen by the agent camera and predicting their properties more likely fails (e.g., the agent fails in predicting whether a fridge is open when it sees only a corner of the fridge).

	Success ↑	*SPL* ↑
Random	1.72%	1.33%
DD-PPO	35.11%	17.37%
DD-PPO$_{boost}$	36.61%	17.49%
OGAMUS	**56.78%**	**24.87%**

Table 6.2: Performance of OGAMUS w.r.t. the random baseline, DD-PPO, and DD-PPO$_{boost}$, evaluated on the object goal navigation task in the ROBOTHOR simulator.

6.3.2 Comparison on Object Goal Navigation

We did not find other approaches using simulator ITHOR that solve the tasks considered in our experiments. Therefore, in our experimental analysis we considered a second simulator, ROBOTHOR [27], for which the last challenge concerning the object goal navigation was launched in 2021.

For the object goal navigation task, we compared OGAMUS with a random baseline, an RL baseline provided in the challenge, called DD-PPO, and the winner of the challenge, called DD-PPO$_{boost}$. Both the RL baseline and the winner exploit the DD-PPO algorithm [118] where the hidden state is computed by providing, as input to a GRU [20], the visual features of the RGB-D images computed by a ResNet-18 [49]. The baseline and the winner approach have been trained on 108,000 episodes for 300 and about 10 million steps, respectively.

For this experiment, we adopt an additional metric, called Success weighted by Path Length (SPL), and introduced in [5]. This metric measures the efficiency of the agent in reaching the goals and is defined as:

$$SPL = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(s_i \cdot \frac{p_i^\star}{max(p_i, p_i^\star)} \right)$$

where N is the number of episodes, $p_i^\star$ is the shortest-path distance from the initial position of the agent to the closest goal in the i-th episode, p_i is the length of the agent path in the i-th episode, and s_i is a boolean variable equal to 1 when the i-th episode succeeds, and equal to 0 otherwise. If the path of the agent is the shortest one, the term in parenthesis is 1. The longer the path, the lower the term in parenthesis and the worse the metric.

For the experiment, we considered the validation episode dataset provided in the challenge, which is composed of 1800 episodes set in the 15 validation scenes of ROBOTHOR. We did not consider the test episode dataset of the challenge, because for such a dataset the evaluation can be done only by the organizers of the challenge who require that the evaluated approach plays by the challenge rule. This is not the case for OGAMUS because it allows the agent to perceive its pose, which is not available in the challenge. While the usage of this additional information can in principle favors OGAMUS w.r.t. the approaches that took part in the challenge, it is worth noting that the agent position can be approximately derived from the RGB-D egocentric views by means

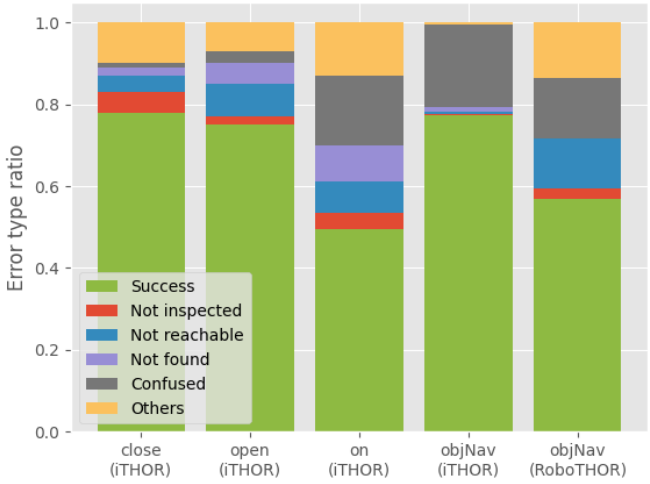


Figure 6.5: Ratio of the occurrences of different error types made by OGAMUS.

of visual simultaneous localization and mapping methods [112]. Most importantly, the usage of the validation dataset of ROBOTHOR disfavors OGAMUS w.r.t. the other compared approaches because the object detector and predicate classifiers of OGAMUS are trained using the training and validation scenes of a different simulation environment, ITHOR, while the other compared approaches are trained and validated on the training and validation scenes of ROBOTHOR.

Each episode of the dataset consists of 500 steps, and regards finding and moving toward objects of 12 types. We trained an object detector similarly to the one for ITHOR simulator, but focused on the 12 goal object types of ROBOTHOR, which provides a performance slightly higher than the object detector trained using all the 118 object types of ITHOR, obtaining 59.02% precision and 69.06% recall.

Table 6.2 shows the results of the comparison. The random baseline provides poor performance. This indicates that, for the ROBOTHOR simulator, the object goal navigation task is quite challenging. The complexity of the task is confirmed by the performance of the RL baseline which is higher than the random baseline but still quite low. DD-PPO$_{boost}$ provides results slightly higher than the RL baseline. Remarkably, OGAMUS outperforms DD-PPO$_{boost}$ in terms of success rate and *SPL*. This confirms that the integration of symbolic planning with state recognition from sensory data can provide competitive results w.r.t. RL approaches.

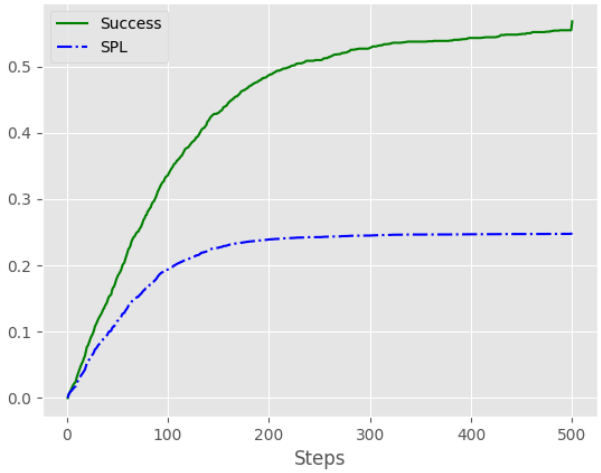


Figure 6.6: Average performance of OGAMUS for the goal object navigation task in the ROBOTHORsimulator, using a number of steps ranging from 0 to 500.

6.3.3 Error Analysis

In Figure 6.5, we analyze the errors made by OGAMUS on all tasks. For few episodes, denoted as "Not inspected", the agent detects a far object of the same type as the type used for the goal definition, and subsequently approaches the object but is no more able to recognize it. This is due to the fact that either the object does not really exist, or the agent does not recognize an existing object, despite being close to and looking at it. For some episodes, namely "Not reachable", the agent finds a goal object but cannot reach a position close enough to the object. This can be due to the fact that either the agent collides or the goal object's estimated position is farther than the real one. Collisions more often happen for the task ON, when the agent holds an object as the agent encumbrance increases. An error in the estimation of the object position is more likely for large objects, such as tables or televisions, since the agent considers the center of the object as its position. There are few episodes, labeled as "Not found", where the agent does not find the object, due to either an ineffective exploration of the environment or false negatives of the object detector. We observed that the latter case is more likely than the former, because the agent almost always explores the entire environment within the given number of steps. The errors labeled as "Confused" denote episodes for which the agent believes it succeeded while the task has not been completed. This is due to false positives of the object detector. Finally, "Others" denote all other task-dependent failures. E.g., for the ON, OPEN, and CLOSE tasks, the agent sometimes fails to identify the object position when it has to manipulate an object. This more likely happens for small objects, such as spoons or saltshakers. Moreover, for the ON

predicate an agent can fail to put an object on a table due to the fact that the target position is already occupied, or there is not enough space on the table.

Figure 6.6 shows the success rate and *SPL* for a number of steps ranging from 0 to 500. For almost all episodes the agent achieves the goal in 300 steps. For a few episodes, the agent achieves the goal only after 500 steps. This happens because the agent is actually close to and looks at a goal object, but it fails to recognize the object.

Chapter 7

Planning for Learning Object Properties

Agents embedded in a physical environment, like autonomous robots, need the ability to perceive objects in the environment and recognize their properties. For instance, a robot operating in an indoor environment should be able to recognize whether a certain box found in the environment is open or closed, a cup is full (of coffee) or empty, the TV is on or off, and so on. From these perceptions, the agent can build and use abstract representations of the states of the environment to reach its goals through automatic planning techniques. The common approach to provide an agent with such perceptual capabilities consists in pre-training offline a (set of) perception models from hundreds of thousands of semantically annotated data (e.g., images or other sensory data). See for instance [6, 53, 28].

In offline training approaches, the perception capabilities are fixed once and for all. This is in stark contrast with a main requirement in many robotics applications: agents embedded in real-world, open-ended environments should be able to dynamically and autonomously improve their perceptual abilities by actively exploring their environments. This is also in agreement with the emerging popular research area of interactive perception [11]. When perception functions are modeled by (deep) neural networks, an open and interesting challenge is whether agents can autonomously decide when and how to improve their perception models, by collecting the needed training data and using them to train the neural network.

We explore a way to address this challenge with an automated planning approach. In particular, we design a PDDL planning domain for *planning to learn (or improve) the perceptual capabilities of the agent*. We focus on the problem of automatically training neural networks able to recognize properties of objects, e.g. open/closed, by relying on a pre-trained object detector. We extend a PDDL planning domain, called *base domain*, with new actions and predicates for learning properties of object types. Such an extension, called

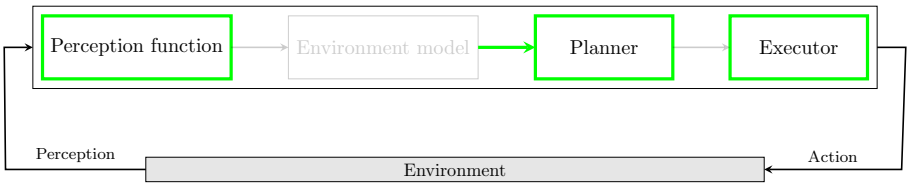


Figure 7.1: OLOP within the agent environment interface.

learning domain, is specified in a meta-language of the base language. It contains the reification of properties and types of the base language. For instance, if `Is_Open` is a property of the base domain, the learning domain contains the object 'is_open' of type `Property`. Furthermore, the learning domain contains actions for collecting training examples for properties, and actions for training the network with them.

The online learning of object properties is obtained by planning in the union of the base and learning domains, and by executing the generated plan. The base domain allows the agent to plan to reach a state where the agent can observe an object with a certain property, e.g., a state where $\texttt{Is\_Open}(\texttt{box}_0)$ is true. The learning domain allows the agent to plan for actions that collect observations of objects with the property being true, e.g., take pictures of $\texttt{box}_0$, which is known to be open, and add them to the positive training examples for the property `Is_Open`. In this way, the agent automatically maps low-level perceptions (e.g., images of an open box) into the symbolic property of objects at the abstract planning level (e.g., "the box is open" in PDDL).

OLOP, in our agent-environment interface (Figure 7.1), learns the perception function, since it plans for collecting data that are used to train object property predictors. Similarly to OGAMUS, the environment model is not learned since a symbolic planning domain is given as input. OLOPheavily relies on the planner, since it plans for learning object properties, and exploits the executor for executing symbolic actions through its actuators.

7.1 Preliminaries and Problem Definition

Perception functions The agent perceives the environment by sensors that return real-value measurements on some portion of the environment. For example, the perceptions of an agent with an onboard camera and a system for estimating its position is a vector (x, y, z) of coordinates and an RGB-D image i taken by the agent's on-board camera. Observations are partial (e.g., the camera provides only the front view) and could be incorrect (e.g, the estimation of the position could be noisy). We suppose that, at the time when a perception occurs, the agent's knowledge about the environment is represented by a grounded planning domain $\mathcal{M}(\mathcal{C})$, where C represents the set of objects already discovered in the environment. Each $c \in \mathcal{C}$ is associated with an anchor [21] that describes the perceptual features of c that have been collected by the agent

so far (e.g., the pictures of c from different angles, the estimated position, and size of c, etc.). At the beginning, the set $\mathcal{C}$ of constants is empty. The agent performs and processes each perception in order to extract some knowledge about the objects in the environment, and about their properties in the current state. This is achieved by combining an *object detector* and a set of *property classifiers*.

The object detector identifies a set of objects in the current perception (e.g., RGB-D image) and predicts their types (i.e., it selects one type among the object types of the planning domain). Every detected object is associated with numeric features (e.g. the bounding box, the estimation of the position, etc.), which are used to build the anchors of the detected object. The features of each detected object are compared with the features of the objects already known by the agent, i.e., those present in the current set of constants $\mathcal{C}$. If the features of the detected object match (to a certain degree) the features of a $c \in \mathcal{C}$, then the features of c are updated with the new discovered features. Otherwise, $\mathcal{C}$ is extended with a new constant c anchored to the features of the detected object, and the type $t(c)$ is asserted in the planning domain, where t is the type returned by the object detector.

For every object c of type t returned by the object detector and for every property p that applies to t, a classifier $\rho_{t,p}$ predicts if c has/has not the property p. Notice that, not all properties apply to a type, e.g., it does not make sense to check if a laptop is filled or empty. Furthermore, for the same property we use different classifiers for different types, since predicting that a bottle is open or that a book is open from visual features are two very different tasks. $\rho_{t,p}$ can be specified either explicitly by a set of predefined rules, or it can be a machine learning model trainable by supervised examples. For instance, the classifier that checks if an object is `Close_To` the agent is defined by a threshold on the distance between the agent and the object position. Other properties (e.g. `Is_Open`) are predicted using a neural network, which takes as input object images and returns the probability of the property being true.

Plan execution To achieve its goal (expressed in a formula of the language of the planning domain), the agent generates a plan using a classical planner (e.g., we used Fast-Forward [51]), and then it executes the plan. However, the symbolic actions of the plan need to be translated into sequences of *operations* executable by the agent's actuators (e.g., rotate of 30°, grasp the object in position x, y, z, move forward of 30cm). Designing effective and robust methods for producing this mapping is a research area that goes out of the scope of this paper, see for instance [32]. In our experiments, we adopt state-of-the-art path planning algorithms (based on a map learned online by the agent) and ad-hoc compilations of actions. However, it is worth noting that we do not assume the execution of the actions leads to the symbolic state predicted by the planning domain. For instance, the execution of the action `Go_Close_To`(c) might end up in a situation where the agent is not close enough to the object c and the predicate `Close_To`(c) is false, despite being a positive effect of the action `Go_Close_To`(c). Moreover, the execution of a symbolic action can have effects

that are not predicted by the action schema. For instance, some properties of an object might become true even if they are not in the positive effects of the symbolic actions. For these reasons, after action executions, the agent must check if the plan is still valid, and if not, it should react to the unexpected situation, e.g., by replanning.

Problem We place an agent at a random position in an unknown environment; we initialize it with the following components: (i) a set of sensors on the environment; (ii) a trained object detector ρ_o; (iii) a planning domain $\mathcal{M} = (\mathcal{P}, \mathcal{O}, \mathcal{H})$; (iv) a method for executing its ground actions; (v) *an untrained neural network* $\rho_{t,p}$ for predicting the property p of the objects of type t, for a subset of pairs (t, p) of interest.

We focus on the online training of the $\rho_{t,p}$'s; our aim is to design a general method to autonomously generate symbolic plans for producing a training set $T_{t,p}$, for every pair (t, p) of interest, and use $T_{t,p}$ to train the perception function $\rho_{t,p}$.

$T_{t,p}$ contains pairs (c, v), where c is (the name of) an object of type t with the associated anchor (e.g., the visual features of the object) and $v \in \{p, \neg p\}$ is the value of the property p. Since $T_{t,p}$ is automatically created by acting in the environment, it may contain wrong labels. We evaluate the effectiveness of our method on the performance (precision and recall) of each $\rho_{t,p}$ against a ground truth data set collected independently by the agent.

7.2 The Proposed Method

We explain the proposed method with a simple example. Suppose an agent aims to learn to recognize the property `Is_Turned_On` for objects of type `Tv`, it can proceed as follows: *(i)* look for an object (say $\texttt{tv}_0$) of type `Tv`; *(ii)* turn $\texttt{tv}_0$ on to make sure that $\texttt{Is\_Turned\_On}(\texttt{tv}_0)$ is true, *(iii)* take pictures of $\texttt{tv}_0$ from several perspectives, and label them as positive examples for `Is_Turned_On`. To produce negative examples for the same property, the agent can proceed in the same fashion, applying the action $\texttt{Turn\_Off}(\texttt{tv}_0)$.

The behavior explained above should be automatically produced and executed by the agent for every learnable pair (t, p), where t denotes an object type and p a learnable property. Therefore, in the following, we explain a procedure that extends automatically the planning domain of the agent to express the goal of learning p for t, and such that the procedure for collecting training data for $\rho_{t,p}$ is generated by a symbolic planner, and can be executed by the agent. This method requires that, for every learnable pair (t, p), the planning domain contains at least an operator applicable to objects of type t that makes p true, and one that makes p false.

This means that we have to extend the planning domain with the capability of expressing facts about its properties and types, i.e., we have to extend it with meta predicates and names for the elements of the planning domain $\mathcal{M}$.

`Observe`(o, t, p):	
pre:	$\neg$`Viewed`(o, t, p)
	`Closed_To`(o)
	`Known`(o, t, p)
eff$^+$:	`Sufficient_Obs`(t, p)
	`Viewed`(o, t, p)
`Explore_For`(t, p)	
pre:	$\forall x$(`Known`$(x, t, p) \rightarrow$ `Viewed`(x, t, p))
eff$^+$:	`Explored_For`(t)
`Train`(t, p, q):	
pre	`Sufficient_Obs`(t, p)
	`Sufficient_Obs`(t, q)
eff$^+$:	`Learned`(t, p, q)

Table 7.1: Schemas for `Observe`, `Search_For` and `Train`.

7.2.1 Extended Planning Domain for Learning

Table 7.1 summarizes how we extend the planning domain for observing, exploring, and learning.

Names for types and properties For each object type $t \in \mathcal{P}$ (e.g. `Box`), we add a new constant 't' (e.g '`box`')[1]. For each object property $p \in \mathcal{P}$ (e.g., `Is_Open`), we add two new constants, namely 'p' and '`not_`p', (e.g., '`is_open`' and '`not_is_open`').

Epistemic predicates We extend $\mathcal{P}$ with predicates for stating that an agent knows/believes that an object has a certain property in a given state. The binary predicate `Known`$(o,$'p'$)$ (resp. `Known`$(o,$'`not_`p'$)$) indicates that the agent knows that the object o has (resp. does not have) the property p. The atom `Known`$(x,$'p'$)$ is automatically added to the positive (resp. negative) effects of all the actions that have $p(x)$ in their positive (resp. negative) effects; similarly, the atom `Known`$(x,$'`not_`p'$)$ is automatically added to the positive (resp. negative) effects of all the actions that have $p(x)$ in their negative (resp. positive) effects. For example, the atoms `Known`$(x,$ '`is_turned_on`') and `Known`$(x,$ '`not_is_turned_on`') are added to the positive and negative effects of `Turn_On(`x`)`, respectively. Similarly, the atoms `Known`$(x,$ '`is_turned_on`') and `Known`$(x,$ '`not_is_turned_on`') are respectively added to the negative and positive effects of `Turn_Off(`x`)`.

Predicates and Operators for Observations We extend the planning domain with the operator `Observe`(o, t, p), which takes as input an object o, a type t, and a property p. The low level execution of `Observe`(o, t, p) consists in extending the training dataset $T_{t,p}$ with observations (i.e. images) of object o taken from different perspectives. The positive effects of `Observe`(o, t, p) contain

[1]Quotes are used to indicate names for elements of $\mathcal{P}$.

the atom $\texttt{Viewed}(o, p)$, and the preconditions of $\texttt{Observe}(o,t,p)$ contain the atom $\neg\texttt{Viewed}(o, p)$, which prevents the agent from again observing o for the property p in the future.

The atom $\texttt{Sufficient\_Obs}(t, p)$ is added to the positive effects of the action $\texttt{Observe}(o, t, p)$. Whether the agent, after executing $\texttt{Observe}(o, t, p)$, has not collected enough observations of objects of type t with property p, the atom $\texttt{Sufficient\_Obs}(t, p)$ is actually false, in contrast with what is predicted by the planning domain, and the agent has to plan for observing other objects of type t.

Predicates and Operators for Exploration The planning domain is extended with the binary operator $\texttt{Explore\_For}(t, p)$ that explores the environment looking for new objects of type t. The precondition of $\texttt{Explore\_For}(t, p)$ is that all the known objects of type t has been viewed for the property p ,i.e., $\forall x(\texttt{Knows}(x, t, p) \rightarrow \texttt{Viewed}(x, t, p))$. Indeed, finding a new object creates a new object o in the planning domain. The effect of $\texttt{Explore\_For}(t, p)$ is $\texttt{Explored\_For}(t)$, which indicates that a new object of type t has been found. $\texttt{Explored\_For}(t)$ is a positive effect of $\texttt{Explore\_For}(t, p)$ in the planning domain. However, the actual execution of $\texttt{Explore\_For}(t, p)$ will not make it true until all the environment has been explored, or a maximum number of iterations has been reached.

Predicates and Operators for Learning We extend the planning domain with the predicate $\texttt{Learned}(t, p, \texttt{not\_}p)$ that becomes true when the agent has collected enough observations, and $\rho_{t,p}$ is trained with them. We add to the planning domain the operator $\texttt{Train}(t, p, q)$. When the agent executes the action $\texttt{Train}(t, p, q)$, the network $\rho_{t,p}$ is trained using $T_{t,p}$ with positive examples and $T_{t,q}$ with negative examples. The preconditions of this action include $\texttt{Sufficient\_Obs}(t, p)$ and $\texttt{Sufficient\_Obs}(t, q)$ that guarantee to have a sufficient number of positive and negative examples for training $\rho_{t,p}$. This action has only one positive effect, which is $\texttt{Learned}(t, p, q)$.

Specifying the goal formula In the extended planning domain, the goal formula g for learning a property p for an object type t is defined as:

$$g = \texttt{Learned}(t, p, \texttt{not\_}p) \vee \texttt{Explored\_For}(t). \tag{7.1}$$

For example, suppose that an agent aims to learn the property `Turned_On` for objects of type `Tv`, then $g =$ `Learned('tv', 'turned_on', 'not_turned_on')`$\vee$`Found_New('tv')`. If the current set of constants contains an object, say $\texttt{tv}_0$, of type `Tv` such that `Viewed(`$\texttt{tv}_0$`,'tv','is_turned_on')` and `Viewed(`$\texttt{tv}_0$`,'tv','not_is_turned_on')` are both false, then the goal is reachable by the plan:

`Go_Close_To(tv`$_0$`)`
`Turn_On(tv`$_0$`)`
`Observe(tv`$_0$`, 'tv','turned_on')`
`Turn_Off(tv`$_0$`)`
`Observe(tv`$_0$`,'tv', 'not_turned_on')`
`Train('tv', 'turned_on','not_turned_on').`

After the execution of all the actions but the last one of the above plan, if the agent has not collected enough training data for '`turned_on`' and '`not_turned_on`', the atoms `Sufficient_Obs('tv','turned_on')` and `Sufficient_Obs('tv','not_turned_on')` will be false, and the last action of the plan cannot be executed. In such a case, the agent has to replan in order to find another tv which has not been observed yet.

Finally, notice that whether all the TVs known by the agent have been observed for the property `Turned_On`, then the formula $\forall x(\texttt{Known}(x,\text{'}\texttt{tv}\text{'},\text{'}\texttt{turned\_on}\text{'}) \rightarrow \texttt{Viewed}(x,\text{'}\texttt{tv}\text{'},\text{'}\texttt{turned\_on}\text{'})$ is true, and the goal can be achieved by generating a plan that satisfies `Found_New('tv')` , i.e., by executing the action `Explore_For('tv','turned_on')`, which explores the environment for new TVs.

Algorithm 4 PLAN AND ACT TO LEARN OBJECT PROPS

Require: $\mathcal{M} = (\mathcal{P}, \mathcal{O}, \mathcal{H})$ a planning domain
Require: $g = \bigwedge_{(t,p)\in TP}(\texttt{Learned}(t,p) \vee \texttt{Explored\_For}(t))$
1: extend $\mathcal{M}$ with actions and predicates for learning
2: $\mathcal{C} \leftarrow$ names for types and properties in $\mathcal{P}$
3: $s \leftarrow \emptyset$
4: $T_{TP} \leftarrow \{T_{t,p} = \emptyset \mid (t,p) \in TP\}$
5: $\rho_{TP} \leftarrow \{\rho_{t,p} = \text{random init.} \mid (t,p) \in TP\}$
6: $\pi \leftarrow \textsc{Plan}(\mathcal{M}(\mathcal{C}), s, g)$
7: **while** $\pi \neq \langle\rangle$ **do**
8: $\quad op \leftarrow \textsc{Pop}(\pi)$
9: $\quad s \leftarrow s \cup \mathsf{eff}^+(op) \setminus \mathsf{eff}^-(op)$
10: $\quad \mathcal{C}, T_{TP}, \rho_{TP} \leftarrow \textsc{Execute}(op)$
11: $\quad s \leftarrow \textsc{Observe}()$
12: $\quad \pi \leftarrow \textsc{Plan}(\mathcal{M}(\mathcal{C}), s, g)$
13: **end while**

Main control cycle The main control cycle of the agent is described in Algorithm 4, which takes as input a planning domain $\mathcal{M}$ and the goal g for learning a set TP of type-property pairs. At the beginning, the set of constants $\mathcal{C}$ contains only the names for types and properties, and the state s is empty (lines 2–3). For every pair $(t,p) \in TP$, the algorithm initializes the training set $T_{t,p}$ to the empty set, and the neural networks $\rho_{t,p}$ (lines 4–5). Then, a plan π is generated (line 6). In the while loop (lines 7–13), the state s is updated according to the action schema (line 9). Next, the first action of the plan is executed and the set of known constants $\mathcal{C}$, the datasets $T_{t,p}$, and the neural networks

$\rho_{t,p}$ are updated (line 10). Notice that, since the perceived effects of action execution might not be consistent with those contained in the action schema, sensing using the not trainable perception functions is necessary, and the state is updated accordingly (line 11). Moreover, since π might be no more valid in the updated state, a new plan must be generated (line 12). The algorithm terminates if either the whole environment has been explored or a maximum number of iterations has been reached, since, in such cases, the atom `Explored_For`(t) is set to true, and plan π for g is empty.

7.3 Experimental Analysis

We evaluate our approach on the task of collecting a dataset and training a set of neural networks to predict the four properties `Is_Open`, `Dirty`, `Toggled`, and `Filled` on 32 object types, resulting in 38 pairs (t, p), since not all properties are applicable to all object types.

Simulated environment We experiment with our approach in the ITHOR [64] photo-realistic simulator of four types of indoor environments. Each environment is a room of one of the following types: kitchen, living room, bedroom, and bathroom. ITHOR simulates a robotic agent that navigates the environment and interacts with the objects by changing their properties (e.g., opening a box, or turning on a tv). The agent has two sensors: a position sensor and an onboard RGB-D camera. For our experiment, we split the 120 different environments, provided by ITHOR, into 80 for training, 20 for validation, and 20 for testing. Testing environments are evenly distributed among the 4 room types.

Object detector For the object detector ρ_o, we used the YoloV5 model [33], which takes as input an RGB image and returns the object types and bounding boxes detected in the input image. For training ρ_o, we have generated the training (and validation) sets by randomly navigating in the training (and validation) environments, and using the ground truth object types and bounding boxes provided by ITHOR. The training and validation sets contain 115 object types and are composed of 259859 and 56190 examples, respectively. For validating the object detector, we performed 300 runs (with 10 epochs for each run) of the genetic algorithm proposed in [33].

Property predictors For the perception functions $\rho_{t,p}$ predicting properties we adopted a ResNet-18 model [49] with an additional fully connected linear layer, which takes as input the RGB image of the object and returns the probability of p being true for the object. We consider that the input object has the property p if the probability is higher than a given threshold (set to 0.5 in our experiments).

Evaluation metrics and ground truth We evaluate each trained $\rho_{t,p}$ using precision and recall against a ground truth dataset $G_{t,p}$. This dataset is obtained directly from the simulator by randomly navigating the 20 testing environments. To obtain the ground truth information, we have used the object annotations provided by ITHOR. For each property, we generated a balanced test set of positive and negative examples. In particular, for the `Is_Open` property we generated a test set with 8751 examples, 2512 for the `Toggled` property, 1310 for the `Filled` property, and 3304 for the `Dirty` one. It is worth noting that the size of the test set for the `Is_Open` property is higher than other ones since the number of object types that can be opened is higher than the ones with other properties. For a similar reason, the size of the test set for the `Filled` property is the lowest one.

7.3.1 Experiments in Simulated Environments

We run our approach in each tested environment for training the neural network model associated to $\rho_{t,p}$ with the training set $T_{t,p}$ generated online. At each run, the agent starts in a random position of the environment and executes 2000 iterations, where at each iteration a low-level operation (e.g. move forward of $30cm$) is executed.

To understand how the errors of the object detector affect the performance, we propose two variants of our approach, namely ND (Noisy Detections) and GTD (Ground Truth Detections). In both variants, the agent trains $\rho_{t,p}$ on the training set $T_{t,p}$ collected in a single environment, and is evaluated on the test set $G_{t,p}$ previously generated in the same environment. In the ND variant, the agent is provided with a pre-trained object detector ρ_o; while in the GTD variant the agent is provided with a perfect ρ_o, i.e., the ground truth object detections provided by ITHOR. In both variants, the neural networks $\rho_{t,p}$'s are trained for 10 epochs with $1e^{-4}$ learning rate; the other hyperparameters are set to the default values provided by PyTorch1.9 [87].

Experimental results We compare the versions ND and GTD for each learned property, the results are shown in Table 7.2. In particular, the columns of Table 7.2 contain the object type, the number of examples collected in the training and test sets, respectively $G_{t,p}$ and $T_{t,p}$, the metrics Precision and Recall averaged over all 20 environments. It is worth noting that the size of the test set can vary among ND and GTD, since we remove from the test set the object types that are missing in the training set, i.e. the object types that have not been observed by the agent. This is because we are interested in evaluating the learning performance on the object types that the agent actually manipulates and observes. Moreover, there are particular object types (e.g. desktop and showerhead in Table 7.2) that are never recognized by the object detector, hence they are missing in the training set, and they are assigned the '-' value in Table 7.2.

In Table 7.2 are shown the results obtained for learning properties `Dirty`, `Filled`, `Is_Open`, and `Toggled`. Not surprisingly, both the weighted average preci-

	size of $G_{t,p}$		size of $T_{t,p}$		Precision		Recall	
Object type	ND	GTD	ND	GTD	ND	GTD	ND	GTD
			Dirty					
bed	564	564	1502	671	0.95	0.57	0.43	0.61
bowl	280	280	383	1027	0.67	0.98	0.81	0.73
cloth	96	210	61	503	0.93	0.95	0.78	0.7
cup	96	262	146	986	0.63	0.99	0.95	0.54
mirror	654	678	2490	3100	0.91	0.9	0.68	0.8
mug	230	432	225	1367	0.88	0.94	0.42	0.74
pan	140	200	20	476	0.76	0.99	0.87	0.79
plate	166	406	47	1304	0.61	0.97	0.97	0.77
pot	210	272	51	929	0.76	0.99	0.91	0.98
Weighted avg	-	-	-	-	0.84	0.89	0.68	0.74
			Filled					
bottle	22	22	78	150	0.65	0	1	0
bowl	328	256	390	1091	0.64	1	0.73	0.77
cup	116	286	200	1028	0.92	0.9	0.56	0.68
houseplant	34	34	18	72	0.5	0.5	0.65	0.82
kettle	-	84	-	337	-	0.25	-	0.4
mug	126	354	250	1136	0.8	0.86	0.51	0.56
pot	226	274	93	809	0.67	1	0.89	0.79
Weighted avg	-	-	-	-	0.7	0.86	0.72	0.66
			Is_Open					
book	148	268	367	1471	1	0.94	0.76	0.81
box	204	204	959	1044	0.92	0.88	0.37	0.54
cabinet	2892	2892	1545	1669	0.81	0.8	0.74	0.79
drawer	3343	3747	1237	2624	0.79	0.75	0.77	0.71
fridge	400	400	803	1109	0.78	0.81	0.72	0.75
laptop	360	360	1124	1531	0.93	0.97	0.85	0.82
microwave	250	250	742	843	0.68	0.82	0.5	0.68
showercurtain	144	134	271	567	0.47	0.96	0.41	0.76
showerdoor	74	140	56	346	0.88	0.71	0.19	0.98
toilet	356	356	1024	1148	0.89	0.9	0.63	0.74
Weighted avg	-	-	-	-	0.81	0.8	0.72	0.75
			Toggled					
candle	54	124	3	118	0.59	0.33	0.63	0.6
cellphone	-	216	-	682	-	0.84	-	0.94
coffeemachine	320	320	999	996	0.95	0.97	0.72	0.61
desklamp	12	56	254	255	1	0.91	1	0.97
desktop	-	56	-	184	-	1	-	0.93
faucet	602	480	921	1663	0.84	0.85	0.89	0.92
floorlamp	44	12	88	68	0.83	0.75	0.5	1
laptop	432	432	1545	1777	0.91	0.83	0.61	0.74
microwave	252	252	1131	1124	1	1	0.76	0.72
showerhead	-	46	-	12	-	1	-	1
television	222	238	269	510	0.99	0.94	0.85	0.95
toaster	280	280	713	1072	0.86	0.98	0.59	0.7
Weighted avg	-	-	-	-	0.9	0.88	0.74	0.8

Table 7.2: Size of the ground truth test set $G_{t,p}$, the generated training set $T_{t,p}$, and performance in terms of precision and recall on the 38 type-property pairs.

sion and recall of the GTD version are almost always higher than the ND ones, i.e. the overall learning performance is better when the agent is provided with ground truth object detections. The recall is generally lower than the precision, this is because, for almost all object types, the number of negative examples is higher than the positive ones, i.e. the training datasets are not balanced. Therefore, the agent is more likely to predict that a property is false, which causes more false negatives and a decrease in the recall. In our experiments, we tried to balance the observations of each object type in the collected dataset by randomly removing positive or negative examples, but we obtained worse learning performance. A more sophisticated strategy might apply criteria to measure the information of each observation and remove the less informative ones, however, we did not tackle this problem since it is out of the scope of this

Figure 7.2: Pepper taking images of a laptop and asking a human to manipulate it for learning the property `Folded`.

paper.

In the `Dirty` property results, for all object types but bed, the number of examples in the training set is higher for GTD, as expected. The examples of objects of type bed in GTD are lower because in all of the environments where there are objects of type bed (i.e. the bedrooms), the agent focused on manipulating and observing objects of types different from bed. Indeed, for all other object types contained in bedrooms (i.e. cloth, mug and mirror), the number of training examples collected by the GTD version is higher than the ND one.

Moreover, for the `Dirty` property, the precision obtained by the GTD version is significantly higher than the ND one for almost all object types (i.e. 7 out of 9). For the mirror object type, the precision achieved by both ND and GTD versions is almost equal. Remarkably, for the bed object type, the precision of the GTD version is much lower than the ND one. This is because, for large objects such as beds, the GTD version is more likely to collect examples not representative for the properties to be learned. For instance, the agent provided with ground truth object detections recognizes the bed even when it sees just a corner of the bed, whose image is not significant for predicting whether the bed is dirty or not. Moreover, the examples of objects of type bed in the training set collected by ND are much higher than the GTD one.

The recall of the GTD version is not always higher than the ND one. In our experiments, we noticed that, for both ND and GTD, an high precision typically entails a low recall, and vice versa. This is because typically the agent collects more positive or negative examples of a single object type. For instance, the precision achieved by ND on object types bed, cloth, mirror, and mug is high and the recall is low. Similarly, the recall achieved by ND on object types bowl, cup, pan, plate, and pot is high and the precision is low. The recall obtained by GTD is lower than the precision for all object types but bed, where there is no significant difference. Overall, the weighted average metric values show good performance, i.e., our approach is effective for learning to recognize properties without any dataset given a priori as input. Similar considerations given for the `Dirty` property apply to results obtained for properties `Is_Open`, `Toggled` and `Filled`, reported in Table 7.2. However, it is worth noting that for the `Filled` property, the metric values obtained by both ND and GTD versions are particularly low

Object type	Property	Precision	Recall
bowl	`Empty`	0.63	0.98
laptop	`Folded`	0.97	1.00
book	`Is_Open`	1.00	0.99
cup	`Filled`	0.93	0.83
Weighted avg	-	0.88	0.95

Table 7.3: Precision and recall obtained by the neural networks predicting object properties in a real environment.

for the object types houseplant and kettle. This is because, for the mentioned object types, the `Filled` property is hard to recognize from the object images. For instance, the fact that an object of type kettle is filled with water cannot be recognized from its image, since the water in the kettle is not visible from an external view such as the agent one. Furthermore, GTD with the object type bottle achieves 0 value of both precision and recall, this is a particular situation where the neural network associated with the `Filled` property never predicts false positives when evaluated on examples of objects of type bottle, hence precision and recall equals 0.

7.3.2 Real World Demonstrator

To test our method in a real-world setting, we used a Softbank's Pepper humanoid robot in PEIS home ecology [99], shown in Figure 7.2. As an object detector, we deployed a publicly available model of YoloV5 pre-trained on the MS-COCO dataset [76]. For manipulation actions, Pepper asks a human to do the manipulations, due to its limited capabilities in manipulating objects. We used Pepper's speech-to-text engine for simple verbal interaction with the human. Given an object type and a property, Pepper first looks for the object and then asks the human about the property's state. Next, it collects samples and asks the human to change the state of the property, and after human confirmation, it further collects samples.

We run experiments for learning pairs type-property reported Table 7.3. For each pair, we run the experiment 7 times, each time with a different setup (e.g. different objects of the same type). At each run, Pepper collects 100 images of the observed property, divided into 50 positive and 50 negative samples. For each object property, we took 4 runs for training (i.e. 400 samples), and 3 runs for testing (i.e. 300 samples). Table 7.3 shows the precision and recall obtained on the test sets. Both the average precision and recall are high. For the simpler properties (i.e. `Is_Open` and `Filled`), Pepper almost perfectly learned to recognize them. These results demonstrate that the proposed approach can be effective also when applied in a real-world environment.

Chapter 8

Online Learning of Reusable Abstract Models for Object Goal Navigation

In Embodied AI, the agent's intelligence emerges from the interaction with the environment as the result of sensorimotor activities [107]. While acting in a real environment, an agent should acquire and effectively represent some knowledge of its surrounding, obtained through sensors (such as an RGB camera).

However, this knowledge acquisition task is challenging and can be accomplished by adopting two main approaches. On the one hand, knowledge can be embedded in a sub-symbolic model (e.g., a neural network). Such a model can be learned (or trained) by means of supervised or RL techniques, which can be directly applied to the sensory data [34, 118]. On the other hand, one can adopt a symbolic representation of the environment, which captures the high-level and relevant aspects of the environment, abstracting away information that is not necessary for the achievement of the agent's goals.

In this Chapter, we follow the second approach, aiming to obtain a more abstract and general knowledge representation that can be, eventually, reused by the agent in the future. To this end, the agent, such as a robot navigating in a complex environment, represents the acquired knowledge of the environment in an *abstract model* that encodes the following key features: (*i*) semantic information about the objects in the environment and their properties; e.g., an agent's state is represented as "the agent is close to a fridge and a table is visible from the current agent's position" (see Fig. 8.1); (*ii*) the elements of the *abstract model* are "grounded" to the perceptions; for instance, the agent stores in the *abstract model* some information about each discovered object, such as the object position, visual features, etc.; (*iii*) the *abstract model* is built online, and incorporates the additional information the agent acquires while acting in the environment; e.g., new objects discovered by the agent while navigating in the environment are added to the *abstract model*; (*iv*) the *abstract model* learned

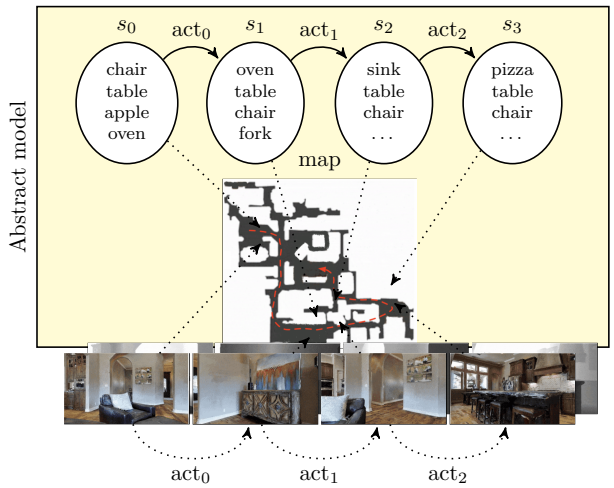


Figure 8.1: An agent navigating in a complex 3D environment. The agent incrementally acquires knowledge about the environment by storing semantic information in an abstract model. For instance, when the agent is in state s_0, objects chair and table are visible; by performing action act_0, other objects become visible, thus the abstract model is updated with the new state s_1, the transition from s_0 and s_1 and the topological map of the environment.

in the past should be reusable by the agent whenever it recognizes to be in an environment that corresponds to a previously learned model. This reusability property is essential because it allows to observe the usefulness of the learned *abstract model* when the agent performs a sequence of episodic tasks in the same environment.

We propose an approach for online learning of reusable abstract models. With respect to our agent-environment interface (Figure 8.2), the proposed approach is focused on learning the environment model and does not assume a perfect perception function is given as input, but rather abstract the perceptions through a noisy perception function. Furthermore, we do not assume high-level actions can be automatically executed through low-level actuators, since the actions executable by the agent are directly low-level navigation actions.

We specifically focus on the Object Goal Navigation (OGN) task [10], where an agent placed in an unknown environment is asked to find and go close to an object of a given goal type. Recent approaches often tackle this problem by building semantic maps of the environments [18, 14] and exploiting SLAM [108, 19] techniques. Instead, we propose to acquire and store the agent's knowledge about the environment in an abstract, and semantically rich, model. Concretely, the learned abstract model is represented by a finite automaton whose set of states explicitly describes what an agent views in different poses. In particular, each state is associated with an agent pose, a set of object types visible

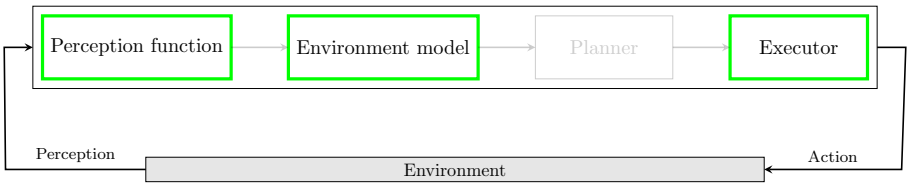


Figure 8.2: The agent environment interface of the proposed approach for online learning of reusable abstract models.

in the agent's view, and the estimated position of each object. We incrementally learn (online) the abstract model by navigating the environment, similarly to [17, 18]. The learned *abstract model* is then stored for future reuse. The information associated with each abstract model state is obtained from low-level perceptions, i.e. a position and orientation sensor, and an RGB and depth camera. When an agent recognizes that its current environment has been already visited before, the agent reuses the previously learned abstract model of the environment, and updates the reused abstract model with the new observations. For recognizing whether the current environment has previously been visited, we design a mechanism that allows the agent to match its current state with the states of different abstract models. We evaluate our approach on the Habitat simulator [101], with 3D real environments provided by the MatterPort3D dataset [16]. Our experiments on the OGN task show that reusing the abstract model is helpful to improve the agent performance over a sequence of episodic tasks (e.g., by improving the optimality of the planned paths).

Summing up, the contributions of the work described in this Chapter are threefold: (*i*) the proposed framework allows an agent to incrementally enhance and reuse previously acquired knowledge, relevant to the current environment; (*ii*) we integrate sub-symbolic techniques, such as semantic segmentation and deep RL, with symbolic reasoning on abstract models; (*iii*) our experimental analysis shows that learning and reusing Abstract Models is an effective way to exploit previously acquired knowledge, obtained from noisy observations (e.g. noisy object detections), for solving the OGN task.

8.1 Object Goal Navigation

In the OGN task [100], an agent is required to go close to an object of a given type (such as *fridge* or *bed*) – referred to as *object goal* – starting from a random position in an unknown and static environment, within a maximum number of actions (set to 500 in our experiments). To reach a goal object, the agent is allowed to execute a set of actions: `move_forward` (by $25cm$), `turn_left`, `turn_right` (by 30°), `stop`. At every step, the agent executes an action and observes the environment via a set of sensors providing an RGB-D image and the agent pose $\langle x, y, \theta \rangle$, relative to the initial one, which is equal to $\langle 0, 0, 0 \rangle$. A single OGN task problem is called an episode. The agent ends an episode by executing

the `stop` action. At the end of each episode, if the distance between the agent position and the closest goal object position is less than a given threshold (set to $1m$ in our experiments), then the episode succeeds; otherwise the episode fails. Solving the OGN task involves multiple challenges. Firstly, the agent has to explore the environment in an effective way, by exploiting SLAM techniques to learn the topological map of the environment. Secondly, the agent has to recognize new objects in the environment whenever they are visible in its current view, by means of, e.g., pre-trained object detection models. Finally, it has to be able to approach the goal object, by means of path planning algorithms to decide which navigation actions to execute.

In the standard OGN task, each episode is independent of the other, and no information is transferred across episodes. We refer to the standard OGN task as *memory-less* setting. We also introduce the *with-memory* setting, where the agent can exploit the knowledge acquired in previous episodes. In the *with-memory* setting, if the agent realizes that is navigating an already visited environment, it can reuse the previously learned abstract model of the environment. We believe that the *with-memory* setting is much closer to real scenarios, where an agent should accumulate and reuse previously acquired knowledge. It is worth noting that the *with-memory* setting introduces new challenges, concerning how and which previously acquired knowledge can be reused in the current situation. For example, matching states of different abstract models, or merging two different abstract models. Furthermore, in the *with-memory* setting, dealing with previously acquired noisy knowledge is even more challenging, due to error accumulation over episodes.

Abstract Model The abstract model of an environment is defined as a finite state machine $\mathcal{D} = \langle S, A, \delta \rangle$ where S is a finite set of states, A is the set of actions executable by the agent, and $\delta : S \times A \to S$ is a deterministic transition function describing transitions between states caused by actions. A state $s \in S$ is represented by a triple $\langle \mathcal{F}_s, \mathcal{C}_s, \{\mathcal{F}_{s,c}\}_{c \in \mathcal{C}_s} \rangle$ where $\mathcal{F}_s$ is a set of continuous features associated with the state (i.e. the visual features extracted from the RGB image of the agent view); $\mathcal{C}_s$ is a finite set of constants representing the objects visible by the agent in s; and, for each $c \in \mathcal{C}_s, \mathcal{F}_{s,c}$ is a set of continuous features associated with object c in s (e.g. the estimated position of c).

Since the agent is aware of different environments, it keeps track of multiple abstract models $\mathcal{D}^{(1)}, \ldots, \mathcal{D}^{(n)}$. We do not assume a one-to-one correspondence between models and environments, since the agent might associate different models with the same environment. For example, the agent could erroneously build two abstract models for the same environment because, the second time it navigates in the environment, it does not realize that the environment has already been visited.

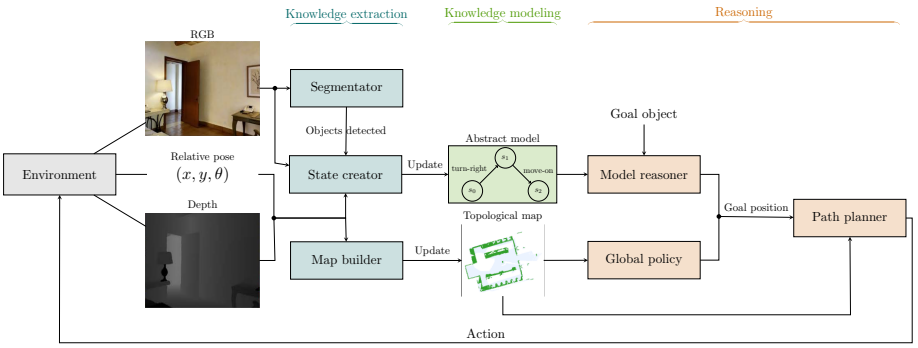


Figure 8.3: Architecture of the proposed approach. The knowledge extraction modules process the sensory data and updates the abstract model and topological map of the environment. The reasoning modules select a goal position on the map to be reached for finding a goal object. Finally, the path planner returns a path plan on the map from the agent position to the goal one.

8.2 Method

An overview of the main cycle executed at every step by the agent for reaching a goal object G is shown in Figure 8.3. The cycle is composed of three main phases: *(i)* knowledge extraction, *(ii)* knowledge modeling, and *(iii)* reasoning. The proposed approach extends [17, 18] by allowing an agent to learn abstract models and reuse them.

Knowledge Extraction. The *segmentator* module [54] extracts object segmentations in the RGB image perceived by the agent. The *map builder* module creates a topological map of the environment with a classical SLAM approach [45] from the current depth image and agent pose. Finally, the *state creator* module generates an abstract state $s = \langle \mathcal{F}_s, \mathcal{C}_s, \{\mathcal{F}_{s,c}\}_{c \in \mathcal{C}_s} \rangle$ where: $\mathcal{F}_s$ is the set of state visual features extracted from the RGB image by an auto-encoder [127]; $\mathcal{C}_s$ are the object types extracted by the *segmentator* module; for all $c \in \mathcal{C}_s, \mathcal{F}_{s,c}$ contains: the position on the map, the bounding box, and the distance from the agent to every visible object of type c. The object position is estimated by adding the depth value of the bounding box centroid to the agent pose.

Knowledge Modeling. In the knowledge modeling phase, the topological map of the environment and the current abstract model are updated with the knowledge extracted in the knowledge extraction phase. Specifically, the current topological map is extended with the additional information available in the agent view, and the current state s, computed by the *state creator*, is added to the abstract model, or updated if already present in the model. Finally, the transition function is extended with (s_{prev}, a, s), where s_{prev} is the previous state, and a is the last executed action.

In the *with-memory* setting, when a state s matches with another state in a previously learned abstract model, the model is reloaded and merged with the current one, according to the procedure described in Section 8.2.1.

Reasoning. In the reasoning phase, given a goal object type g, the agent checks whether the current Abstract Model contains a state with an object of type g (i.e., $\exists s \in S : g \in \mathcal{C}_s$). In such a case, the agent selects one object of type g and computes a path plan from its current position to the object position in the topological map of the environment. If the abstract model contains multiple states with objects of type g, then the agent ranks these objects according to the number of states from which they are visible, and selects the closest one among the five most frequently seen objects. We prefer the most frequently seen objects in order to mitigate the errors of the *segmentator*. Indeed, the more points of view from which an object is detected (i.e. the more states an object belongs to), the less the probability that the object is a false positive detection of the *segmentator*. Alternatively, if the abstract model does not contain any state with objects of type g (e.g. while exploring a new environment, the agent might not have seen any object of type g), a goal position is computed by the *global policy*, which is a deep RL policy proposed in [17, 18]. Given the current topological map, the *global policy* looks for a goal position on the topological map that maximizes the environment exploration. Once the goal position is set, either by the reasoner or by the global policy, the agent computes a plan with a path planner, based on the fast marching algorithm [104], to reach the goal position, and executes the first action in the path plan. To compute a path plan, all unexplored areas of the environment map are considered traversable; this enables the agent to discover new areas of the environment and objects, thus enriching both the environment map and the abstract model.

8.2.1 Abstract Model Reuse

In the *with-memory* setting, the abstract model learned at each episode is stored by the agent for future reuse. Therefore, the knowledge of the agent is constituted by n Abstract Models $\{\mathcal{D}^{(1)}, \dots, \mathcal{D}^{(n)}\}$. Whenever the agent starts a new episode, it initializes a new abstract model $\mathcal{D}^{(n+1)}$. At every step of an episode, the agent looks if its current state $s = \langle \mathcal{F}_s, \mathcal{C}_s, \{\mathcal{F}_{s,c}\}_{c \in \mathcal{C}_s} \rangle$ matches a state in $\{\mathcal{D}^{(1)}, \dots, \mathcal{D}^{(n)}\}$. In particular, for each set of states $S^{(i)} \in \mathcal{D}^{(i)}$, the match between s and a state $s^{(i)} \in S^i$ is performed by means of the cosine distance among the state features:

$$s^* = \underset{\substack{s^{(i)} \in S^{(i)} \\ i \in 1,\dots,n}}{\operatorname{argmin}} \mathit{cos\_dist}(\mathcal{F}_s, \mathcal{F}_{s^{(i)}})$$

When $\mathit{cos\_dist}(s, s^*)$ is lower than a given threshold (set to 0.3 in our experiments), $\mathcal{D}^{(n+1)}$ is merged with $\mathcal{D}^{(i)}$ and the resulting abstract model is considered as the current one. The resulting model contains all the states of the two merged models, and the knowledge is incrementally enhanced through

episodes. After such a merging the agent does not look for further matching in the current episode.

Notice that the matching could not be perfect since the poses of the agent in the matched states s and s^* may be slightly different. This matching difference can propagate to the object positions stored in the abstract model, thus the agent can rely on wrong information. To prevent these potential errors, we propose two different strategies: namely *hard* and *soft*. In the *hard* strategy, we assume that the matching is always perfect and the agent blindly believes in the matched abstract model, i.e., it goes to the goal object position returned by the *model reasoner* without looking for other goal objects on its path. In the *soft* strategy, the agent tries to mitigate the effects of non-perfect matches by looking for the goal object in the area around the goal object position given by the matched abstract model. The dimension of the area around the goal object is proportional to the distance between the agent position in the matching state and the goal object one. Moreover, while navigating an environment, the agent continuously looks for a goal object, possibly terminating the episode before reaching the area around the goal object position provided by the abstract model.

8.3 Experimental Analysis

In our experimental analysis, we evaluate the effectiveness and efficiency of our approach for solving the OGN task. We aim to experimentally show that the *reuse* of previously acquired knowledge, in the form of abstract models, can improve the performance of state-of-the-art approaches for solving the OGN task. Furthermore, we empirically demonstrate our claims with a failure analysis and a qualitative comparison of reusing *vs* not reusing previously acquired knowledge.

8.3.1 Implementation Details

For the experiments, we used the Habitat Simulator [101] with the Matterport3D dataset [16], which contains 90 different scenes (i.e. apartments) with a total of 194000 RGB-D images. Habitat simulates a mobile robot navigating in each of the 90 scenes. The *global policy*, which selects the exploration goal position, is trained for 10 million steps on the 56 training scenes of Matterport3D (specifically 50 training scenes and 6 validation scenes), by means of the Proximal Policy Optimization RL algorithm [118]. The *global policy* architecture is composed of 5 convolutional layers with the ReLU activation functions, and a max pooling layer. For the *semantic segmentation*, we used the RedNet model [54] pre-trained on the 40 object classes available in the training scenes of Matterport3D. The features extractor, which computes $\mathcal{F}_s$ from the RGB data, is the encoder proposed in [127], and $\mathcal{F}_s$ is a vector of dimension 2048. The cosine distance threshold for states matching is set to 0.3. The SLAM algorithm, which computes the topological map of the environment, is based on [45].

Evaluation Metrics The OGN task is evaluated with four standard metrics: the *Success Rate*, the *Success weighted by Path Length (SPL)*, the *SoftSPL*, and the *Distance To Success (DTS)*. All the evaluation metrics but the *SoftSPL* have already been described in Section 6.3. The *SoftSPL* [14] is similar to the *SPL*, but measures the path optimality in all episodes, without penalizing the unsuccessful ones with a zero score; it is defined as:

$$\text{SoftSPL} = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \frac{d_{fin_i}}{d_{init_i}}\right)\frac{p_i^*}{\max(p_i, p_i^*)}$$

where N is the number of episodes, and, for each *i-th* episode, p_i^* is the shortest-path length from the agent's initial position to the closest goal object, p_i is the length of the agent path, d_{init_i} is the distance between the agent initial position and the closest goal object, and d_{fin_i} is the distance between the agent final position and the goal object. It is worth noting that d_{fin_i} equals 0 for successful episodes. With respect to the *SPL*, when an episode fails the *SoftSPL* takes into account the optimality of the path followed by the agent and weights it according to the final distance from the agent to the goal object.

8.3.2 Reusing abstract models

We investigate different ways of reusing knowledge, and the corresponding advantages, by comparing the following four models:

Active Neural SLAM (ANS*): it is our implementation [1] of the ANS model [17] described in Section 3.4; and a baseline for our evaluation, since it does not exploit any previously acquired knowledge.

Hard Pre-explored (ANS*+HP): it is our basic extension of **ANS***, based on the approach proposed in Section 8.2. For each environment, the agent is provided with an input abstract model, which is built by performing 10000 exploration steps; for every episode, the agent can reuse one of the pre-acquired abstract models, by applying the *hard* strategy.

Soft Pre-explored (ANS*+SP): it is similar to **ANS*+HP**; however, the agent reuses the provided abstract models by applying the *soft* strategy.

Soft Incremental (ANS*+SI): the agent is provided with no abstract model as input; afterward, during each episode, the agent can reuse and incrementally extend the abstract models learned in previous episodes, by applying the *soft* strategy.

[1] We checked the coherence of our implementation by running the same experiments as in [17]; The results achieved by our implementation of **ANS** are comparable to the results provided by **ANS** authors in [17]. Specifically, **ANS** achieved 7.056, 0.321 and 0.119 of *DTS*, *success rate*, and *SPL*, respectively; our implementation obtained 6.721, 0.313 and 0.127 on the same metrics.

Method	DTS↓	Success↑	SPL↑	SoftSPL↑
ANS*	6.417	0.240	0.102	0.191
ANS*+HP	6.352	0.251	0.105	0.206
ANS*+SP	6.294	0.258	0.117	0.214
ANS*+SI	**6.155**	**0.279**	**0.131**	**0.233**

Table 8.1: Results achieved by the baseline **ANS*** and our approach variants on the Matterport3D validation set.

The fundamental difference between **ANS*/ANS*+SI** and **ANS*+HP/ANS*+SP** is that the former do not exploit pre-acquired knowledge, while the latter require such knowledge. Moreover, **ANS*+SI** is our only version in which the agent extends the abstract models with the additional knowledge acquired through episodes. Finally, all the versions but **ANS*** use the *with-memory* setting described in Section 8.1.

In Table 8.1, we report the results of our variants on the validation set of Matterport3D, composed of 2195 episodes in 11 environments., that is a standard benchmark for the OGN task [18, 14]. **ANS*+HP** achieves higher results than **ANS***, as expected, since **ANS*+HP** is provided with additional input knowledge. Furthermore, **ANS*+SP** obtains better results than **ANS*+HP**, due to the fact that the *soft* strategy mitigates the errors introduced by the matching of abstract models in different episodes (Section 8.2.1). Remarkably, **ANS*+SI** outperforms all other versions, providing a relative improvement of +8.13% in *success* and +11.9% in *SPL* w.r.t. **ANS*+SP**. The results of Table 8.1 show that the incremental learning of abstract models is more effective than providing the agent with the pre-acquired input abstract models.

Furthermore, the fact that the agent starts in each episode from a different position allows the **ANS*+SI** variant to discover environment areas that, for some large environments, are hardly reachable with a single long pre-exploration. Notably, **ANS*+SI** is able to match states of different abstract models in 69.7% of the episodes.

8.3.3 Effects of Knowledge Accumulation

We experimentally evaluate how accumulating knowledge in the abstract models affects the agent performance for solving the OGN task. To better investigate the usefulness of knowledge accumulation, we limit the quantity of noise recorded in the abstract model (e.g., the false positives given by the semantic segmentator). Therefore, we evaluate our approach on a subset of the Matterport3D validation set, where the *semantic segmentator* achieves better performance. This subset is built as in [18] and contains episodes with the following goal object types: *chair, sofa, plant, bed, toilet, tv, table,* and *sink*.

Table 8.2 reports the results achieved by **ANS*** and **ANS*+SI**. The results show that reusing abstract models (**ANS*+SI**) allows the agent to follow better paths (15% *SPL*) and go closer to the goal objects (6.34m *DTS*), with respect to **ANS***.An analysis of the success rate evolution through episodes is shown

Method	DTS↓	Success↑	SPL↑
ANS*	6.721	0.313	0.127
ANS*+SI	**6.347**	**0.354**	**0.150**

Table 8.2: Results obtained on a subset of the validation set of Matterport3D, containing object types which are in both MS-COCO and MatterPort3D datasets (658 episodes across 11 environments).

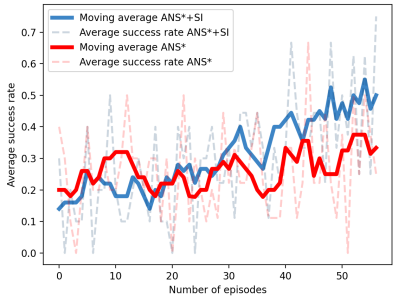


Figure 8.4: The average success rate and moving average success rate achieved by **ANS*** and **ANS*+SI**. The window size of the moving average success rate is equal to 5.

in Figure 8.4. The dashed curves are the success rates achieved by **ANS*** and **ANS*+SI**, during each episode, and averaged over all 11 environments. For example, when the number of episodes equals 0, the reported value is the average success rate across the 11 environments in their first episode. The thick curves represent the moving averages of the success rate with a window size of 5. For example, when the number of episodes equals 10, the reported value is the average success rate across the 11 environments and episodes 8–12. The comparison among the moving averages of **ANS*** and **ANS*+SI**, shown in Figure 8.4, confirms that accumulating knowledge over episodes consistently enhances the success rate.

8.3.4 Semantic Maps and Abstract Models

A different method for representing knowledge about the environments is by means of semantic maps [14]. Semantic maps are topological maps of the environments enriched with information about object types. We compare approaches based on semantic maps with our method and investigate how semantic maps and abstract models can be combined.

A method for solving the OGN task, which exploits an input semantic map, is **SMNet** [14]. In **SMNet**, the plan to reach a goal object is provided by computing the shortest path from the current agent position to the position of

Method	DTS↓	Success↑	SPL↑	SoftSPL↑
SMNet[14]	7.316	0.096	0.057	0.087
SMNet (GT)	**5.658**	0.312	**0.207**	**0.282**
ANS*+SI	6.155	0.279	0.131	0.233
SemExp*+SI	5.785	**0.347**	0.151	0.274

Table 8.3: Results obtained on the validation set of the Matterport3D dataset. Notice that **SMNet (GT)**, as described in [14], exploits input semantic maps with ground truth free areas.

an object of the goal type in the input semantic map. Note that, in **SMNet**, the input semantic map may be incomplete. However, **SMNet** has the simplifying assumption that the absolute position of the agent is known, therefore matching different semantic maps is not necessary. In [14], authors also consider a version of **SMNet**, named **SMNet(GT)**, that assumes ground truth free space maps, i.e., input semantic maps that, despite being possibly incomplete, has the full information about the environment areas traversable by the agent.

Another method that exploits semantic maps is **SemExp** [18], which is based on **ANS**; however, in **SemExp**, the *global policy* takes semantic maps as input, rather than topological maps as in **ANS**. The *global policy* of **SemExp** seeks to directly find the goal object position, instead of finding a position to be reached in order to maximize the environment exploration. For evaluating how semantic maps and abstract models can be combined, we combine our **SI** approach with **SemExp**; this version is called **SemExp*+SI**.

Table 8.3 compares our different versions for incrementally reusing knowledge against **SMNet** and **SMNet(GT)**. We perform the experiment on the same validation set used in [14], which is the benchmark validation set for the OGN task of the Matterport3D dataset. Remarkably, **ANS*+SI** and **SemExp*+SI** outperform **SMNet** by a large margin. Furthermore, the *global policy* adopted in **SemExp*+SI** increases all the metrics w.r.t. **ANS*+SI** policy. The high *success* rate of **SemExp*+SI** w.r.t. **ANS*+SI** (+6.8%) suggests that the way in which the environments are explored plays a crucial role in how the learned abstract models are reusable. Interestingly, **SemExp*+SI** has similar performances to **SMNet (GT)**, despite **SMNet (GT)** exploits input semantic maps with ground truth free space.

8.3.5 Limitations and Failure Analysis

One of the major limitations of our model comes from the abstraction of the agent's perceptions. The output of the *semantic segmentator*, as well as the visual features associated with each state, may be affected by errors. Furthermore, in the *without-memory* setting, the abstract model does not provide a significant added value with respect to simpler representations, such as semantic maps. However, this is not the case in the *with-memory* setting, where the abstraction encoded in the abstract models is a cornerstone for the reuse of

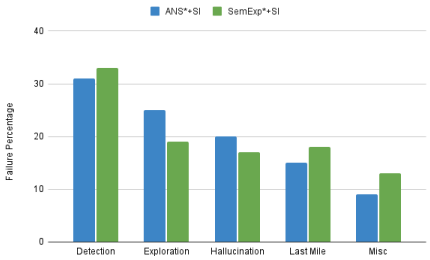


Figure 8.5: Analysis of the failure causes of **ANS*+SI** and **SemExp*+SI** evaluated on the validation set of Matterport3D.

previously acquired knowledge.

In the following, we report a failure analysis for understanding why the agent fails in the **SI** setting. In particular, we quantitatively investigate how the errors introduced by the semantic segmentator affect the reliability of the learned abstract models. To this aim, we randomly sample 200 failed episodes from the experiment in Table 8.2, where a previously learned abstract model has been reused. We group the failure causes into five classes: (*i*) **Last mile (navigation failure):** the agent correctly navigated to an instance of the goal object but was not able to reach it (i.e. the DTS is less than 2 meters but greater than 1 meter); (*ii*) **Hallucination (abstract model failure):** the agent approached the goal position extracted from the abstract model, but there was no goal object nearby the goal position; (*iii*) **Detection (sensors failure):** the agent, during its path to the goal position suggested by the abstract model, found a wrong goal object (i.e. an object of a type different from the goal one), and approached it; (*iv*) **Exploration (abstract model incompleteness):** the reused abstract model had no information about possible goal object positions, and the agent could not find any goal object within 500 steps; (*v*) **Others:** the agent reused the abstract model but had a generic failure (e.g. getting stuck in a corner of a room).

Interestingly, we have two possible failure causes directly linked to the reloaded abstract model: exploration, where the agent has not enough information about the environment, and hallucination, where the agent relies on wrong information. In Figure 8.5 are reported the statistics about the failed episodes. The abstract model generated by **SemExp*+SI** gave fewer failures w.r.t. **ANS*+SI** in the exploration and hallucination classes, highlighting how the different *global policy* affects the creation of the abstract models. Furthermore, the majority of the failures are in the detection class for both models. This suggests that a better semantic segmentator could improve the overall performance by a large margin.

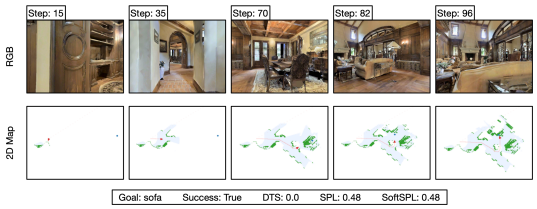


Figure 8.7: A successful episode with the **ANS*+SI** variant. The agent correctly matched its current state with one in a previously learned Abstract Model and exploited the information provided by the reused Abstract Model for successfully navigating towards a goal object sofa.

8.3.6 Qualitative examples

In Figure 8.6 and 8.7, we report a qualitative comparison among **ANS*** and **ANS*+SI** on the same episode in a specific scene of the MatterPort3D dataset.

ANS* starts by exploring the environment, but it never encounters an object of the goal type sofa. Furthermore, the exploration leads the agent very far (i.e. more than 10 meters) from the nearest sofa to the agent's initial position. This is because the environment of the considered scene is very large w.r.t. the average dimension of other environments, and the agent is likely to follow paths towards environment areas that are far from the goal object. Therefore, with a limited number of 500 steps, the agent cannot easily find a goal object, when navigating into the environment for the first time. Figure 8.7 shows the same episode with the exploitation of a previously learned abstract model. Particularly, at step 15, the agent state matches a state of the previously learned abstract model, which is then reused. The agent position is rescaled to match the reused domain one, and the agent selects the goal position provided by the reused abstract model. Afterward, the agent creates an exploration area around the goal position, which it has to explore for finding the goal object. At step 82, the agent reached the exploration area and found the sofa, therefore it finally approached the sofa and the episode ended successfully.

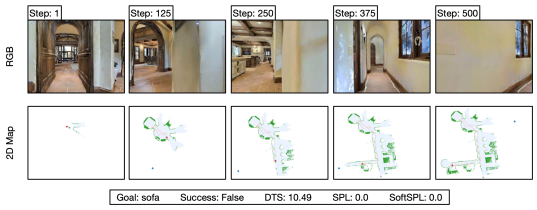


Figure 8.6: A failed episode with the **ANS*** variant. The agent explored the environment for 500 steps without finding the goal object sofa. Green regions on the map represent obstacles, light blue regions are explored areas, and the blue point is the goal position.

Chapter 9

Conclusions and Future Works

The integration of learning and planning for agents acting in unknown environments can be grouped into two main paradigms: learning for planning and planning for learning. Many works in the AI planning literature focus on learning for planning, e.g., most of the work on learning planning domains, or the majority of the approaches that integrate RL and symbolic planning. Less attention has been paid to planning for learning, i.e., to approaches that take advantage of symbolic planning for applying learning methods. One of the main novelties of planning for learning approaches is that they specify the learning problem as a planning problem, where the planning goal defines a set of goal states that allows an agent to gain experience and knowledge useful for learning.

On the learning for planning side, the integration of learning and planning allows to overcome the limitations of planning, which perform reasoning at a high-level of abstraction, assuming an agent can observe its symbolic state without necessarily considering the low-level perceptions given by its sensors. Learning methods allows to drop this simplifying assumption, providing an agent with the capability of linking its state perceived through continuous sensors with its symbolic state. It is worth noting that there is an additional complexity when dealing with symbolic states derived from sensory data: the agent symbolic state is not necessarily correct and complete. For example, a robotic agent provided with an on-board RGB camera has a partial view of the environment, and, since the perception capabilities of the agent are not perfect, the detections of objects in the image may be noisy and incomplete. Moreover, when operating in complex and partially observable environments (e.g. an apartment), it is very hard to provide an agent with a model of the environment that perfectly specifies all possible states and transitions.

On the planning for learning side, most data-driven learning techniques assume a sufficiently large amount of data is given, and learning is performed offline. However, in many real scenarios, such a large amount of data is not

always available a priori, e.g., when the environment is unknown. Exploiting symbolic planning enables an agent to autonomously collect data necessary for applying data-driven learning methods online. This is particularly important for developing an agent that improve its perception capabilities and adapt them to operate in its current environment. Nonetheless, taking advantage of symbolic planning allows an agent to plan for reaching informative states, i.e., states from which the agent can acquire new knowledge about the environment, by perceiving and interacting with the environment.

We have proposed an architecture for integrating learning, planning, and acting. In our approach, an agent maps its perceptions of the environment into a symbolic state, and learns a high-level model of the environment dynamics by executing symbolic actions through its actuators. We firstly focused on the problem of learning an extensional representation of a discrete and deterministic planning domain from continuous perceptions [69]; assuming the capability of executing symbolic actions through actuators was given. Then, we proposed an approach for learning action models under the assumption of full observability [70]. The proposed approach incrementally learns an action model by selecting goals to reach and actions to execute that allow to acquire useful information about the operators. We showed some important theoretical properties of completeness and integrity of the learned models. The proposed approach achieved good learning performance on a large set of benchmarks from the International Planning Competitions (IPCs), and outperformed a state-of-the-art method for learning action models offline. The proposed method works with full observability; extension to partial observability is part of future work. Afterward, we addressed the problem of online grounding of planning domains in unknown environments [71]. The proposed solution enables an agent to map the sensory data into a symbolic state, allowing it to perform and exploit efficient planning in a wide variety of different environments. We have tested the proposed method on different tasks, obtaining better results than RL-based approaches. Future work will focus on learning a policy to compile high-level actions into low-level executable operations. Next, we presented a method that allows an agent to incrementally acquire and store knowledge about a set of unknown environments [13]. Our method reuses the acquired knowledge, represented as an abstract model, when the agent operates in a previously visited environment. We evaluated the proposed method on the object goal navigation task. Our experiments showed that reusing abstract models of previously visited environments can be effective for solving the object goal navigation task. We experimented with different strategies of reusing the acquired knowledge, empirically proving that the abstract models incrementally learned achieve better performance than offline learned ones. Finally, we addressed the challenge of using symbolic planning to automate the process of learning perception capabilities [72]. We focus on learning object properties, assuming a pre-trained object detector is given. We experimentally showed that our approach is feasible and effective for learning the visual properties of objects in both simulated and real environments. Still a lot of work must be done to address the general problem of planning and acting to learn in a physical environment.

Acknowledgements

I thank my (future) wife Benedetta for her endless love, comprehension, patience, and many other things. I probably would not have achieved the requirements for starting a Ph.D. without her. I thank my parents for their support and guidance, I hope to do for them in the future what they have done for me in the past. I also thank my grandparents for always believing in me, I wish they could see me finishing the Ph.D.

I thank Prof. Renata Mansini, which advised me during my bachelor's and master's thesis, and pushed me towards starting a Ph.D., encouraging and helping me. I thank Prof. Luciano Serafini, the mentor of my Ph.D., that spent his time teaching me how to do research, and not only. I still have to learn a lot, but I hope I will be for my future students what Luciano has been for me. I also thank Dr. Paolo Traverso, the Director of Research at FBK, for motivating me during my Ph.D., and being an example of what a real leader is. Paolo and Luciano are giant researchers that showed me the way forward for becoming a good researcher, and not only; I hope I will not lose it in the future. I thank my advisors from the University of Brescia, Prof. Alfonso Gerevini, for guiding me through the Ph.D., and Prof. Alessandro Saetti, for the time spent discussing and all his efforts. Luciano, Paolo, Alfonso, and Alessandro advised me during the Ph.D., if we did good research, I think is mainly thanks to them. I thank Prof. Alessandro Saffiotti for the opportunity to visit the University of Örebro, collaborate with his research group, and for all the support during my visit.

Finally, I thank all my friends from FBK and the University of Brescia, going beyond being colleagues has also strongly improved my daily working life, and has been significantly helpful when going through stressful periods. Similarly, I thank all friends I met during my visit to the University of Örebro; they made my experience unforgettable.

Bibliography

[1] David Abel, Dilip Arumugam, Lucas Lehnert, and Michael L. Littman. State abstractions for lifelong reinforcement learning. In *ICML*, 2018.

[2] David Abel, Nate Umbanhowar, Khimya Khetarpal, Dilip Arumugam, Doina Precup, and Michael Littman. Value preserving state-action abstractions. In *International Conference on Artificial Intelligence and Statistics*, pages 1639–1650. PMLR, 2020.

[3] Alper Ahmetoglu, M Yunus Seker, Justus Piater, Erhan Oztop, and Emre Ugur. Deepsym: Deep symbol generation and rule learning from unsupervised continuous robot interaction for planning. *arXiv preprint arXiv:2012.02532*, 2020.

[4] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia. Learning action models with minimal observability. *Artif. Intell.*, 275:104 – 137, 2019. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2019.05.003.

[5] Peter Anderson, Angel Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, et al. On evaluation of embodied navigation agents. *arXiv preprint arXiv:1807.06757*, 2018.

[6] Masataro Asai. Unsupervised grounding of plannable first-order logic representation from images. In *ICAPS*, 2019.

[7] Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI*, 2018.

[8] Masataro Asai and Christian Muise. Learning neural-symbolic descriptive planning models via cube-space priors: the voyage home (to strips). In *Proceedings of the Twenty-Ninth International Joint Conferences on Artificial Intelligence*, pages 2676–2682, 2021.

[9] Fahiem Bacchus. The AIPS '00 planning competition. *AI Magazine*, 22 (3):47–56, 2001.

[10] Dhruv Batra, Aaron Gokaslan, Aniruddha Kembhavi, Oleksandr Maksymets, Roozbeh Mottaghi, Manolis Savva, Alexander Toshev, and

Erik Wijmans. ObjectNav revisited: On evaluation of embodied agents navigating to objects. *arXiv preprint arXiv:2006.13171*, 2020.

[11] Jeannette Bohg, Karol Hausman, Bharath Sankaran, Oliver Brock, Danica Kragic, Stefan Schaal, and Gaurav Sukhatme. Interactive perception: Leveraging action in perception and perception in action. *IEEE Transactions on Robotics*, 33:1273–1291, 2017.

[12] Blai Bonet and Hector Geffner. Learning first-order symbolic representations for planning from the structure of the state space. In *ECAI*, 2020.

[13] Tommaso Campari, Leonardo Lamanna, Paolo Traverso, Luciano Serafini, and Lamberto Ballan. Online learning of reusable abstract models for object goal navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14870–14879, 2022.

[14] Vincent Cartillier, Zhile Ren, Neha Jain, Stefan Lee, Irfan Essa, and Dhruv Batra. Semantic mapnet: Building allocentric semanticmaps and representations from egocentric views. 2021.

[15] Michal Certicky. Real-time action model learning with online algorithm 3SG. *Applied Artificial Intelligence*, 28(7):690–711, 2014.

[16] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. 2017.

[17] Devendra Singh Chaplot, Dhiraj Gandhi, Saurabh Gupta, Abhinav Gupta, and Ruslan Salakhutdinov. Learning to explore using active neural slam. *arXiv preprint arXiv:2004.05155*, 2020.

[18] Devendra Singh Chaplot, Dhiraj Prakashchand Gandhi, Abhinav Gupta, and Russ R Salakhutdinov. Object goal navigation using goal-oriented semantic exploration. *Advances in Neural Information Processing Systems*, 33:4247–4258, 2020.

[19] Devendra Singh Chaplot, Ruslan Salakhutdinov, Abhinav Gupta, and Saurabh Gupta. Neural topological slam for visual navigation. 2020.

[20] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, 2014.

[21] Silvia Coradeschi and Alessandro Saffiotti. An introduction to the anchoring problem. *Robotics and autonomous systems*, 43(2-3):85–96, 2003.

[22] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[23] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *Twenty-First International Conference on Automated Planning and Scheduling*, 2011.

[24] Stephen Cresswell, Thomas Leo McCluskey, and Margaret Mary West. Acquiring planning domain models using *LOCM*. *Knowledge Eng. Review*, 28(2):195–213, 2013.

[25] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. In *Machine learning techniques for multimedia*, pages 21–49. Springer, 2008.

[26] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2): 1–142, 2013.

[27] Matt Deitke, Winson Han, Alvaro Herrasti, Aniruddha Kembhavi, Eric Kolve, Roozbeh Mottaghi, Jordi Salvador, Dustin Schwenk, Eli VanderBilt, Matthew Wallingford, Luca Weihs, Mark Yatskar, and Ali Farhadi. RoboTHOR: An Open Simulation-to-Real Embodied AI Platform. In *CVPR*, 2020.

[28] Nils Dengler, Tobias Zaenker, Francesco Verdoja, and Maren Bennewitz. Online object-oriented semantic mapping and map updating. In *2021 European Conference on Mobile Robots (ECMR)*, pages 1–7. IEEE, 2021.

[29] Pedro Zuidberg Dos Martires, Nitesh Kumar, Andreas Persson, Amy Loutfi, and Luc De Raedt. Symbolic learning and reasoning with noisy data for probabilistic anchoring. *Frontiers in Robotics and AI*, 7, 2020.

[30] Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. A survey of embodied ai: From simulators to research tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2022.

[31] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: a comprehensive survey and benchmark. *Neurocomputing*, 2022.

[32] Manfred Eppe, PD Nguyen, and Stefan Wermter. From semantics to execution: Integrating action planning with reinforcement learning for robotic tool use. *arXiv preprint arXiv:1905.09683*, 2019.

[33] Glenn Jocher et al. ultralytics/yolov5: v6.0 - YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support, October 2021.

[34] Kuan Fang, Alexander Toshev, Li Fei-Fei, and Silvio Savarese. Scene memory transformer for embodied agents in long-horizon tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 538–547, 2019.

[35] Alan Fern, Sung Wook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *ICAPS*, 2004.

[36] Maria Fox and Derek Long. The 3rd international planning competition: Results and analysis. *CoRR*, abs/1106.5998, 2011.

[37] Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 209–217, 1998.

[38] Ramón García-Martínez and Daniel Borrajo. An integrated approach of learning, planning, and execution. *J. Intell. Robotic Syst.*, 29(1):47–78, 2000.

[39] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[40] Yolanda Gil. Learning new planning operators by exploration and experimentation. In *AAAI*, 1994.

[41] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *ICML*, 1994.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[43] Peter Gregory and Stephen Cresswell. Domain model acquisition in the presence of static relations in the lop system. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 97–105, 2015.

[44] Martin Günther, JR Ruiz-Sarmiento, Cipriano Galindo, Javier González-Jiménez, and Joachim Hertzberg. Context-aware 3d object anchoring for mobile robots. *Robotics and Autonomous Systems*, 110:12–32, 2018.

[45] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. 2017.

[46] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[47] Jessica B Hamrick, Abram L Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.

[48] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[50] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[51] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22 (3):57–57, 2001.

[52] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5): 359–366, 1989.

[53] Michael Janner, Sergey Levine, William T Freeman, Joshua B Tenenbaum, Chelsea Finn, and Jiajun Wu. Reasoning about physical interactions with object-oriented prediction and planning. *arXiv preprint arXiv:1812.10972*, 2018.

[54] Jindong Jiang, Lunan Zheng, Fei Luo, and Zhijun Zhang. Rednet: Residual encoder-decoder network for indoor rgb-d semantic segmentation. *arXiv preprint arXiv:1806.01054*, 2018.

[55] Brendan Juba and Roni Stern. Learning probably approximately complete and safe action models for stochastic worlds. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):9795–9804, 2022.

[56] Brendan Juba, Hai S Le, and Roni Stern. Safe learning of lifted action models. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 18, pages 379–389, 2021.

[57] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996.

[58] Kei Kase, Chris Paxton, Hammad Mazhar, Tetsuya Ogata, and Dieter Fox. Transferable task execution from pixels through deep planning domain learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10459–10465. IEEE, 2020.

[59] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[60] Hiroaki Kitano and Satoshi Tadokoro. Robocup rescue: A grand challenge for multiagent and intelligent systems. *AI Mag.*, 22(1):39–52, 2001.

[61] Andreas Knoblauch, Rebecca Fay, Ulrich Kaufmann, Heiner Markert, and Günther Palm. Associating words to visually recognized objects. In *Anchoring symbols to sensor data. Papers from the AAAI Workshop. Technical Report WS-04-03*, pages 10–16. AAAI Press Menlo Park, 2004.

[62] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[63] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[64] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017.

[65] George Konidaris. On the necessity of abstraction. *Current opinion in behavioral sciences*, 29:1–7, 2019.

[66] George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[67] Roshan Kumari and Saurabh Kr Srivastava. Machine learning: A review on binary classification. *International Journal of Computer Applications*, 160(7), 2017.

[68] Hanard Kurutach, Aviv Tamar, Ge Yang, Stuart Russell, and Pieter Abbeel. Learning plannable representations with causal infogan. In *NIPS*, 2018.

[69] Leonardo Lamanna, Alfonso Emilio Gerevini, Alessandro Saetti, Luciano Serafini, and Paolo Traverso. On-line learning of planning domains from sensor data in pal: Scaling up to large state spaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11862–11869, 2021.

[70] Leonardo Lamanna, Alessandro Saetti, Luciano Serafini, Alfonso Gerevini, and Paolo Traverso. Online learning of action models for pddl planning. In *IJCAI-2021*, 2021.

[71] Leonardo Lamanna, Luciano Serafini, Alessandro Saetti, Alfonso Emilio Gerevini, and Paolo Traverso. Online grounding of symbolic planning domains in unknown environments. In *Proceeding of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022*, 2022.

[72] Leonardo Lamanna, Luciano Serafini, Mohamadreza Faridghasemnia, Alessandro Saffiotti, Alessandro Saetti, Alfonso Gerevini, and Paolo Traverso. Planning for learning object properties. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 2023.

[73] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks: A unified approach to action segmentation. In *European conference on computer vision*, pages 47–54. Springer, 2016.

[74] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for mdps. In *ISAIM*, 2006.

[75] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[76] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[77] Amy Loutfi, Silvia Coradeschi, and Alessandro Saffiotti. Maintaining coherent perceptual information using anchoring. In *IJCAI*, pages 1477–1482, 2005.

[78] D. Lyu, F. Yang, B. Liu, and S. Gustafson. SDRL: Interpretable and data-efficient deep reinforcement learning leveraging symbolic planning. In *AAAI*, 2019.

[79] Zhihao Ma, Yuzheng Zhuang, Paul Weng, Hankz Hankui Zhuo, Dong Li, Wulong Liu, and Jianye Hao. Learning symbolic rules for interpretable deep reinforcement learning. *arXiv preprint arXiv:2103.08228*, 2021.

[80] Dastan Maulud and Adnan M Abdulazeez. A review on linear regression comprehensive in machine learning. *Journal of Applied Science and Technology Trends*, 1(4):140–147, 2020.

[81] Drew McDermott, Malik Ghallab, A. Howe, Craig A. Knoblock, A. Ram, M. Veloso, Daniel S. Weld, and David E. Wilkins. PDDL—The planning domain definition language. Technical Report DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, 1998.

[82] Drew V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

[83] Piotr Mirowski. Learning to navigate. In *1st International Workshop on Multimodal Understanding and Learning for Embodied Applications*, pages 25–25, 2019.

[84] V. Mnih, K. Kavukcuoglu, D. Silver, A.i A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[85] Kira Mourão, Luke S. Zettlemoyer, Ronald P. A. Petrick, and Mark Steedman. Learning STRIPS operators from noisy and incomplete observations. In *UAI*, 2012.

[86] Arsalan Mousavian, Alexander Toshev, Marek Fišer, Jana Košecká, Ayzaan Wahid, and James Davidson. Visual representations for semantic target driven navigation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8846–8852. IEEE, 2019.

[87] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32: 8026–8037, 2019.

[88] Andreas Persson, Pedro Zuidberg Dos Martires, Luc De Raedt, and Amy Loutfi. Semantic relational object tracking. *IEEE Transactions on Cognitive and Developmental Systems*, 12(1):84–97, 2019.

[89] Andreas Persson, Pedro Miguel Zuidberg Dos Martires, Luc De Raedt, and Amy Loutfi. Probanch: a modular probabilistic anchoring framework. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence,*, pages 5285–5287. International Joint Conferences on Artificial Intelligence, 2020.

[90] Arthur George Richards. *Robust constrained model predictive control*. PhD thesis, Massachusetts Institute of Technology, 2005.

[91] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1):107–136, 2006.

[92] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Incremental learning of relational action rules. In *ICMLA*, 2010.

[93] Christophe Rodrigues, Pierre Gérard, Céline Rouveirol, and Henry Soldano. Active learning of relational action models. In *ILP*, 2011.

[94] Ivan D Rodriguez, Blai Bonet, Javier Romero, and Hector Geffner. Learning first-order representations for planning from black box states: New results. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 18, pages 539–548, 2021.

[95] Usha Ruby and Vamsidhar Yendapalli. Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng*, 9(10), 2020.

[96] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[97] Jose-Raul Ruiz-Sarmiento, Martin Günther, Cipriano Galindo, Javier González-Jiménez, and Joachim Hertzberg. Online context-based object recognition for mobile robots. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 247–252. IEEE, 2017.

[98] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[99] Alessandro Saffiotti, Mathias Broxvall, Marco Gritti, Kevin LeBlanc, Robert Lundh, Jayedur Rashid, Beom-Su Seo, and Young-Jo Cho. The peis-ecology project: Vision and results. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2329–2335, 2008. doi: 10.1109/IROS.2008.4650962.

[100] Manolis Savva, Angel X. Chang, Alexey Dosovitskiy, Thomas Funkhouser, and Vladlen Koltun. Minos: Multimodal indoor simulator for navigation in complex environments. *arXiv preprint arXiv:1712.03931*, 2017.

[101] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. 2019.

[102] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[103] Luciano Serafini and Paolo Traverso. Learning abstract planning domains and mappings to real world perceptions. In *AI*IA 2019 - Advances in Artificial Intelligence*, volume 11946 of *Lecture Notes in Computer Science*, pages 461–476. Springer, 2019.

[104] James A Sethian. Fast-marching level-set methods for three-dimensional photolithography development. In *Optical Microlithography IX*, volume 2726, pages 262–272. International Society for Optics and Photonics, 1996.

[105] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[106] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[107] Linda Smith and Michael Gasser. The development of embodied cognition: Six lessons from babies. *Artificial Life*, 11(1–2):13–30, January 2005.

[108] Cyrill Stachniss, John J. Leonard, and Sebastian Thrun. Simultaneous localization and mapping. In *Springer Handbook of Robotics*, Springer Handbooks, pages 1153–1176. Springer, 2016.

[109] Roni Stern and Brendan Juba. Efficient, safe, and probably approximately complete learning of action models. In *26th International Joint Conference on Artificial Intelligence, IJCAI 2017*, pages 4405–4411. International Joint Conferences on Artificial Intelligence, 2017.

[110] R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[111] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

[112] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. Visual slam algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, 9(1):1–11, 2017.

[113] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[114] Thomas J Walsh and Michael L Littman. Efficient learning of action schemas and web-service descriptions. In *AAAI*, 2008.

[115] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2022.

[116] Xuemei Wang. Planning while learning operators. In *AAAI*, 1996.

[117] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[118] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. In *International Conference on Learning Representations*, 2019.

[119] Mitchell Wortsman, Kiana Ehsani, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. Learning to learn how to learn: Self-adaptive visual navigation using meta-learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6750–6759, 2019.

[120] Joseph Z. Xu and John E. Laird. Instance-based online learning of deterministic relational action models. In *AAAI*, 2010.

[121] Joseph Z. Xu and John E. Laird. Combining learned discrete and continuous action models. In *AAAI*, 2011.

[122] Joseph Zhen Ying Xu and John E. Laird. Learning integrated symbolic and continuous action models for continuous domains. In *AAAI*, 2013.

[123] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[124] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171 (2-3):107–143, 2007.

[125] Yuan Yang and Le Song. Learn to explain efficiently via neural logic inductive learning. *arXiv preprint arXiv:1910.02481*, 2019.

[126] Joel Ye, Dhruv Batra, Abhishek Das, and Erik Wijmans. Auxiliary tasks and exploration enable objectnav. *arXiv preprint arXiv:2104.04112*, 2021.

[127] Amir R. Zamir, Alexander Sax, William Shen, Leonidas J. Guibas, Jitendra Malik, and Silvio Savarese. Taskonomy: Disentangling task transfer learning. 2018.

[128] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-model acquisition from noisy plan traces. In *IJCAI*, 2013.

[129] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, 174(18):1540–1569, 2010.