UNIVERSITA’ DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN
MATEMATICA

CICLO XXXV

A Bottom-up Method for Rule Construction in Neural-Symbolic Reinforcement Learning

Coordinatore:
Chiar.mo Prof. Alessandra Lunardi

Tutore:
Chiar.mo Prof. Federico Bergenti

Dottorando: Davide Beretta

Anni Accademici 2019/2020 – 2021/2022

To my parents and to my sister

To my grandparents

To Martina

Abstract (English)

Deep learning has been increasingly successful in the last few years, and it obtained a plethora of impressive results in various contexts. However, the drawbacks and the limitations of deep learning and related approaches have recently become indisputable. Actually, the results of deep learning algorithms are hardly interpretable, and they struggle to generalize to unseen situations. In order to overcome these problems, *neural-symbolic* methods have been recently proposed as a viable approach in various contexts because neural-symbolic methods combine symbolic with sub-symbolic machine learning methods to gain the advantages of both while avoiding that inherent problems. This dissertation focuses on the proposal of a new neural-symbolic method for neural-symbolic reinforcement learning by discussing its design, the developed implementation, and some experimental results. The first part of this dissertation overviews the ordinary reinforcement learning concepts, and it surveys the major deep reinforcement learning and relational reinforcement learning approaches available in the literature. The second part of the dissertation focuses on a structured comparison among some of the most representative neural-symbolic approaches for inductive logic programming, which can be considered as good candidates for neural-symbolic reinforcement learning. The last part of the dissertation presents the proposed method as an evolution of an existing algorithm proposed by Jiang & Luo and called Neural Logic Reinforcement Learning. The original algorithm provides for the generation of rules using a top-down approach, while the algorithm proposed in this dissertation uses a bottom-up approach that generates rules starting from the states of the environment to obtain general rules to be used for training. The proposed algorithm is presented and compared with the original algorithm to empirically assess the validity of the approach. The proposed method is able to effectively learn many tasks and it successfully generalize to slightly different versions of the training tasks. However, despite requiring less information from the user, it performs worse than NLRL. This dissertation is concluded with a discussion on some limitations of the proposed algorithm and with an overview of future research directions.

Abstract (Italian)

Negli ultimi anni, il deep learning ha avuto sempre maggior successo ed ha permesso di ottenere risultati molto significativi in diversi ambiti. Tuttavia, sono sempre più evidenti i limiti di queste tecniche, le quali non sono interpretabili e faticano a gestire situazioni mai viste. Allo scopo di superare i limiti intrinseci del deep learning, i metodi *neuro-simbolici* sono stati recentemente proposti come soluzione in vari contesti perché questi metodi combinano le tecniche di machine learning sub-simbolico con quelle di machine learning simbolico allo scopo di ottenere i vantaggi di entrambe evitandone gli svantaggi. Questo lavoro di tesi di dottorato di ricerca propone un approccio neuro-simbolico per risolvere problemi di reinforcement learning, discutendone la progettazione, la relativa implementazione ed alcuni risultati sperimentali. Nella prima parte di questo lavoro saranno introdotti i principali concetti di reinforcement learning, saranno inoltre presentati i principali approcci di deep reinforcement learning e relational reinforcement learning. La seconda parte di questo lavoro presenta un confronto tra i principali approcci neuro-simbolici basati su inductive logic programming, i quali possono essere considerati una buona base di partenza per risolvere problemi di reinforcement learning. L'ultima parte di questo lavoro presenta una nuova tecnica, la quale è progettata partendo da un algoritmo esistente introdotto da Jiang & Luo nel 2019 e chiamato Neural Logic Reinforcement Learning. Nell'algoritmo originale le regole di partenza vengono generate in modo top-down. Viceversa, in questa nuova proposta si usa un approccio bottom-up. Infatti, l'algoritmo proposto genera le regole partendo dagli stati forniti dall'ambiente allo scopo di ottenere regole generali da utilizzare durante la fase di addestramento. La nuova tecnica viene presentata e confrontata con quella originale, allo scopo di mostrarne empiricamente la validità. Il metodo proposto è in grado di completare efficacemente diversi task e riesce a generalizzare a versioni leggeremente diverse dei task usati per l'addestramento. Tuttavia, pur richiedendo meno informazioni da parte dell'utente, ottiene risultati peggiori rispetto a NLRL. La parte conclusiva di questo lavoro di tesi di dottorato di ricerca discute i limiti di questo nuovo approccio e presenta una panoramica su alcuni sviluppi futuri.

Contents

List of Figures

List of Tables

1 Introduction **1**

2 Reinforcement Learning **7**

2.1 Basic RL concepts . . . 8
2.1.1 Finite MDP . . . 9
2.1.2 Reward . . . 11
2.1.3 Episode . . . 11
2.1.4 Policy . . . 12
2.1.5 Value function . . . 13
2.1.6 Applications . . . 15
2.2 RL techniques . . . 16
2.2.1 Dynamic Programming . . . 16
2.2.2 Monte Carlo . . . 18
2.2.3 Temporal-Difference . . . 19
2.2.4 Policy Gradient . . . 21

3 Deep and Relational Reinforcement Learning **25**

3.1 Deep RL . . . 26
3.1.1 DQN . . . 26
3.1.2 A3C . . . 27
3.1.3 ACER . . . 28
3.1.4 PPO . . . 31
3.2 Relational RL . . . 31
3.3 Discussion . . . 34

4 A Survey of Methods for ILP 37
4.1 Background 37
4.1.1 First-order logic 39
4.1.2 ILP 40
4.2 Neural-symbolic ILP 41
4.2.1 δILP 42
4.2.2 NTPs 48
4.2.3 ILP_{Camp} 49
4.2.4 dNL-ILP 51
4.2.5 DLM 53
4.2.6 Meta_{Abd} 56
4.3 Comparing methods from the XAI perspective 58
4.3.1 Language 59
4.3.2 Search method 59
4.3.3 Recursion 60
4.3.4 Predicate invention 61
4.4 Comparing Datalog-based methods 62
4.4.1 Representation of data 63
4.4.2 Language bias 63
4.5 Open challenges 64

5 The Proposed Method SD-NLRL 67
5.1 Motivation 68
5.2 NLRL 68
5.3 RL tasks 75
5.4 Inducing rules from states 77
5.5 Inducing general rules from states 83
5.6 Learning to avoid an action 93
5.7 Dealing with recursive patterns within states 96
5.8 Discussion 100

Conclusions 103

Bibliography 107

List of Figures

2.1 Agent-environment interaction in MDPs 10

3.1 DQN neural network architecture 26

5.1 Visual representation of block manipulation and cliff-walking 76
5.2 Average returns and losses for the CLIFFWALKING task 91
5.3 Average returns and losses for the WINDYCLIFFWALKING task 92
5.4 Average returns and losses for the STACK task 100
5.5 Average returns and losses for the UNSTACK task 101
5.6 Average returns and losses for the ON task 102

List of Tables

4.1 Comparison of neural-symbolic ILP methods from the perspective of rule interpretability 60
4.2 Comparison of neural-symbolic ILP methods from the perspective of rule reusability 62

5.1 NLRL hyper-parameters 87
5.2 A performance comparison between NLRL and SD-NLRL on cliff-walking tasks 90
5.3 The rules generated by the third prototype of SD-NLRL 94
5.4 A performance comparison between the second prototype of SD-NLRL and the third prototype of SD-NLRL on cliff-walking tasks 95
5.5 A performance comparison between NLRL and SD-NLRL on block manipulation tasks 99

List of Algorithms

1 The function to transform a ground atom into an atom containing variables 82

2 The function to transform a ground rule into a rule containing variables . 83

3 The function that performs rule generation 84

4 The function that performs rule generation considering also recursive patterns 97

Chapter 1

Introduction

In recent years *Reinforcement Learning* (*RL*) [1] has received more and more interest from the research community. RL is an area of machine learning whose goal is to train an agent to take actions that maximize some form of cumulative reward. RL differs from the other traditional areas of machine learning, supervised and unsupervised, which require a dataset that must be given before the training phase. Actually, RL agents try to learn how to interact with an environment that changes as a consequence of the their actions. RL is not specifically designed for specific tasks, and it is an interesting approach for many real problems.

RL has been studied for many decades now, but, recently, the combinations of these techniques with modern deep learning methods gave birth to *Deep Reinforcement Learning* (*DRL*) [2]. Combining RL with deep learning allows to tackle many real-world problems that require the agent to cope with an uncertain environment. DRL has proved to be an effective approach for many RL tasks, and the obtained results in the last few years are impressive. The first work that represents a milestone for DRL is the paper published by DeepMind in 2013 [3], which presents a method called DQN. DQN reached impressive results in many different tasks, and it represents a turning point for the entire RL field. Other notable DRL works are Alpha-Go [4] and Alpha-Go Zero [5], which were both released after DQN. They defeated a professional player on Go, a game that is considered very difficult due to its large search space. The final goal of the research in the RL field is to design an agent that is able to learn different tasks with a single training. Recent developments in the DRL field follow this direction. Some examples are the works proposed by OpenAI and DeepMind to solve many different games for the Atari 2600 consoles reaching super-human scores [3, 6].

Despite the relevant improvements, the limits of DRL has become more and more evident in the last few years. In particular, these technologies produce solutions that

are not interpretable by humans. Moreover, these techniques are typically not able to generalize to unseen situations. The main goal of this dissertation is to study the limits of current approaches to RL, and to introduce a new RL method that tries to overcome the limitations of current RL methods. This dissertation started from analyzing DRL methods, and the goal was to understand how the limits of DRL, in terms of interpretability and generalization capability, could be tackled. The first step in this research was the study of modern DRL techniques. From this study it was clear that deep neural networks do not represent a solution to the discussed problems but their ability to treat noisy and erroneous data is very useful in real applications. The next step was to study the symbolic approaches to RL. In particular, this dissertation focuses on *Relational Reinforcement Learning* (*RRL*) [7], which has been studied for more than 20 years now. RRL tries to apply symbolic technologies, like logical trees, to RL. This study suggested that RRL approaches produce interpretable solutions that are able to generalize to unseen states. Moreover, RRL approaches are very data-efficient because they require only a small number of interactions to reach reasonable performance. However, RRL methods are not typically good at treating noisy and erroneous data, and they struggle to handle complex tasks. The study performed on RRL and DRL suggested that the best approach would be to combine the ideas behind both these technologies.

In the last few years, neural-symbolic approaches have gained the attention of an increasing number of researchers. These methods combine sub-symbolic algorithms, like deep neural networks, with symbolic ones, trying to obtain the best from both these approaches. The research on this topic started only a couple of years ago, but it could represent the bridge between RRL and DRL. For these and analogous reasons, it was then natural to focus on neural-symbolic methods. In particular, the next step was to understand the state of the art of neural-symbolic methods to *Inductive Logic Programming* (*ILP*) [8] problems. ILP is a research topic focused on the induction of logical rules from data. Neural-symbolic methods for ILP are particularly interesting because they can be easily adapted to deal with RL problems. Current neural-symbolic methods for RL are very limited, and they require the user to guide the search for a solution specifying many hyper-parameters or defining the structure of the resulting program [9–11]. The last step of the research discussed in this dissertation was to design a new RL method, and in particular a new neural-symbolic method, for RL that prevents the user from specifying too much information about the solution of a task while, at same time, producing interpretable solutions. In fact, this requirement represents an important limitation of current neural-symbolic methods for RL. The proposed algorithm is based on *Neural Logic Reinforcement Learning* (*NLRL*) [9], a recent neural-symbolic method that tries to

induce logic rules to solve RL tasks.

The research contributions proposed by this dissertation are two. The first contribution is a comparison of the main neural-symbolic methods to ILP that have been proposed in the last five years. The second contribution is a new neural-symbolic method for RL problems that tries to induce logic rules directly from the states of the environment. From the induced rules, the algorithm tries to obtain more general rules that can be used to tackle slightly different versions of the same task.

The contents of this dissertation is described below. Apart from this introductory chapter, this dissertation is organized as follows.

Chapter 2 presents an introduction to RL, and it describes the basic concepts such as rewards and observations. Moreover, the chapter presents a brief description of the most interesting applications of RL. Finally, the chapter is concluded with an introduction on the main approaches that can be used to tackle RL problems, such as Monte Carlo.

Chapter 3 introduces DRL, and it presents the most important works on the subject of the last few years, such as DQN [3,12], A3C [13], ACER [14], and PPO [6]. Then, the chapter introduces RRL, and it discusses the history of this research subject from the very first work proposed by Džeroski in 1998 [7]. In order to give the reader an overview of the history of RRL, the chapter then presents the most interesting developments made in first years of research. Finally, the chapter is concluded with a discussion on the limits of both DRL and RRL.

Chapter 4 presents a comparison of the main neural-symbolic methods for ILP. As mentioned above, neural-symbolic methods has been chosen to overcome the limits of both RRL and DRL. In particular, neural-symbolic ILP represents an interesting starting point to deal with RL problems. This chapter starts introducing the basic concepts of first order logic, and it then presents a formal definition of inductive logic programming. In order to give a broad overview on this subject, the most representative neural-symbolic methods for ILP of the last five years are described. In particular, the algorithms that has been taken into consideration are six: δILP [15], NTPs [16–18], dNL-ILP [11,19,20], ILP_{Camp} [21], DLM [10], and Meta_{Abd} [22]. Each method is presented, and it is briefly analyzed highlighting the main problems in terms of performance and user guidance, whenever possible. The core of this chapter are two comparisons of the introduced methods. The first comparison is made from the perspective of the interpretability of the

induced rules. It takes into consideration five methods that are all based on program templates or meta-rules: δILP, NTPs, ILP_{Camp}, dNL-ILP, and Meta_{Abd}. The algorithms are compared using four characteristics: the language used to represent the induced rules, the chosen search method, the support of recursive rules, and the support of predicate invention. The second comparison is made from the perspective of the reusability of the induced rules. In particular, the three methods that are taken into consideration, δILP, dNL-ILP and Meta_{Abd}, induce logic rules written in a Datalog dialect. The algorithms are compared using two characteristics: the representation of training examples and background knowledge, and the language bias that is used for rules generation. Both these comparisons are also discussed in two papers presented in 2022 at international events (EXTRAAMAS and LPNMR) [23, 24]. The chapter is concluded presenting some open challenges for neural-symbolic ILP, suggesting possible improvements and interesting research directions.

Chapter 5 introduces a new neural-symbolic algorithm for RL. It is based on NLRL, another neural-symbolic RL method that is in turn an adaptation of δILP for RL tasks. The first part of this chapter presents NLRL, discussing its major limitations. Then, the new method is presented, discussing the main modifications, and the tests made to measure the overall performance of the proposed method. In order to perform a fair comparison among the approaches, the performance of the new proposal is measured on the same tasks discussed in the paper that presents NLRL. In particular, the considered tasks are: CLIFFWALKING, WINDYCLIFFWALKING, STACK, UNSTACK and ON. The first two tasks are very similar, and the agent must learn to reach a goal position in a grid, avoiding specific cells that represent a cliff. The other three tasks require the agent to manipulate blocks. In particular, the ON task requires the agent to stack two specific blocks, STACK requires the agent to stack all the blocks on a single pile, while UNSTACK requires the agent to put all the blocks on the floor. In order to improve the performance of the new algorithm, different modifications has been tested. Here, four versions of the same algorithm are discussed. The first version simply transforms each state in a corresponding rule, replacing constants with variables. The second version makes use of the rules that are generated after the abstraction procedure. The third version adds an unreachable rule for each action, which could be useful when a valid rule that implements the action is not available. The fourth version of the algorithm employs an advanced abstraction procedure that handle recursive patterns within states. The four versions are compared with NLRL on the discussed tasks, and the chapter is concluded with a brief summary of the main obtained results.

Chapter 6 concludes this dissertation, and it summarizes the results of this research. Moreover, this chapter discusses some open challenges in the field and the weaknesses of the proposed method. Finally, a brief summary of future research directions and possible improvements of the proposed method concludes this dissertation.

Chapter 2

Reinforcement Learning

Learning through interactions with an environment is probably the most natural form of learning. An example is an infant, which must learn to move in the world using its own body. As a child grows up, it always learns by interacting with the surrounding environment. For example, driving a car o using a computer are activities that require the user to perform actions and observe the result, in order to learn the proper behaviour to complete the task. This form of learning is the core idea of *Reinforcement Learning* (*RL*) [1]. RL has many interactions with other fields such as optimization and statistics. Moreover, the connections with psychology and neuroscience are very strong. RL can be considered the most similar form of learning to the one used by humans and animals. It is not a coincidence that some historical RL algorithms are inspired by biological beings.

This chapter is structured as follows. Section 1 describes the core concepts of RL, it then continues defining a RL problem and some theoretical solutions such as Bellman optimality equations. In particular, Section 1 defines Finite MDP, which is a theoretical framework that models a decision-making process. Then, it defines the concept of reward, which is the value that is used by the agent to learn which action should be taken in a specific situation. Then, it defines the episode, which is used to break the interaction between the agent and the environment into a sub-sequence of interactions that ends with a terminal state. Then, it presents the concepts of policy and value function, which define how the agent behave and how the agent values a specific situation, respectively. Finally, it concludes with some examples. Section 2 presents the main approaches that can used to tackle RL problems.

2.1 Basic RL concepts

In RL, the goal is to learn how to interact with an environment to maximize a numerical reward value. The agent typically starts interacting with no information about the correct action to take in a specific situation, and it tries to discover which actions lead to the maximum reward. In order to learn the correct behaviour, the agent tries an action and it observes the resulting immediate reward. In complex situations, learning from immediate rewards is not sufficient, and the agent must take into consideration the possible subsequent states and rewards to maximize the cumulative reward in the long run. Trial-and-error and delayed rewards are the core characteristics of RL.

RL is different from the other classic forms of machine learning. In supervised learning, the user provides a set of examples of the desired behaviour to be learned. This requirement is difficult to satisfy in problems in which the agent must interact with an environment, because it is often impossible to provide such examples for all possible interactions. At the same time, RL is different from unsupervised learning. In fact, finding hidden patterns in collections of unlabeled data, which is the goal of unsupervised learning, can be useful in RL but it is not sufficient to solve the problem of maximizing a reward value in the long run.

A core challenge in RL is to manage the trade-off between exploration and exploitation. The information that the agent collects during training is used to increase the obtained reward. Therefore, the agent must exploit the experience made in the past to improve its performance. However, exploiting the accumulated experience too much can be detrimental because, in many situations, trying something new can lead to better results. Therefore, the agent must also explore by trying actions that have not been selected before in the current state. Both exploration and exploitation are important, an agent should try different actions and take advantage of the learned information to make better decisions as the learning progresses. Real problems typically present a stochastic behaviour, and the agent must try each state-action pair several times to obtain a reliable estimate of the expected reward. Finding an optimal trade-off between exploration and exploitation is very difficult, and it is a problem that still remains unsolved.

A RL agent is not necessarily only an isolated agent that interact with an uncertain environment. An RL agent can be part of a larger system. In this case, the agent interacts directly with the rest of the system, and indirectly with the environment. Learning by interacting with larger systems, and only indirectly with the environment, is another important feature of RL.

2.1.1 Finite MDP

RL problems are typically formalized using a *Markov Decision Process* (*MDP*). MDPs represent a way to mathematically model a decision-making problem where the outcomes are only partially under the control of the agent. In particular, in MDPs the agent must be able to access the state of the environment. Moreover, the agent can take actions that could modify the state. Finally, the agent must have one or more goals regarding the state. MDPs model these three aspects: perception, action, and goal. A RL method represents an effective way to solve problems that can be formalized using MDPs. In most real applications, the agent cannot directly access the state of the environment. In this case, the problem can be formalized using a *Partially Observable Markov Decision Process* (*POMDP*). POMDPs are a generalization of MDPs in which the state is not directly available to the agent, and the underlying dynamics of the environment are determined by an MDP. The agent must perceive the state, and the obtained observation is not necessarily equal to the underlying state of the MDP.

Now, a formal description of MDPs is presented. As already mentioned, in MDPs there are two major entities: agent and environment. The environment represents everything that is considered external to the agent. The agent represents the learner that takes actions, interacting with the environment at discrete time steps $t \in \mathbb{N}$. Basically, an MDPs is defined by $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p \rangle$, where:

- $\mathcal{S} \neq \varnothing$ is the set of possible states;
- $\mathcal{A} \neq \varnothing$ is the set of possible actions;
- $\mathcal{R} \subseteq \mathbb{R}$ is the set of possible rewards; and
- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the *dynamics function*.

Note that $\mathcal{A}(s)$ is more precise because the action space depends on the current state but $\mathcal{A}$ is used instead to simplify the notation. The agent interacts with the environment receiving, at each time step t, a representation of the state $s_t \in \mathcal{S}$. Using the obtained state representation, the agent takes and action $a_t \in \mathcal{A}$. Then, at the $t + 1$ step, the agent receives the state s_{t+1} and a reward value $r_{t+1} \in \mathcal{R}$. Figure 2.1 illustrates the agent-environment interaction.

Let R_t, S_t and A_t be random variables representing the possible reward, state, and action at time step t, respectively. Finite MDPs are a class of MDPs that are particularly important to the theory of RL. In finite MDPs, the sets $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ are finite. Therefore, R_t and S_t have a well defined discrete probability distribution that depends only on the

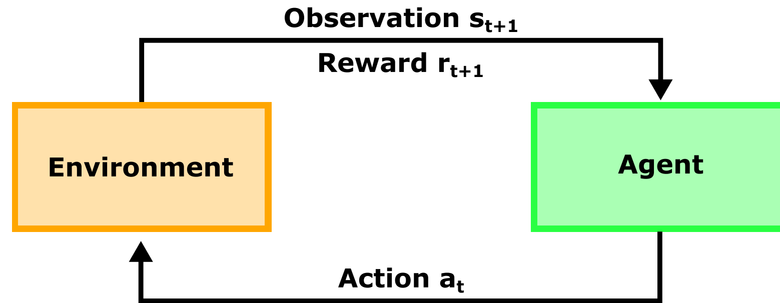

Figure 2.1: Agent-environment interaction in a MDP.

preceding state-action pair. A fundamental characteristic of finite MDPs is the dynamics function p, which is defined as:

$$p(s', r|s, a) = P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

for all $s, s' \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p completely defines the dynamics of the environment. In particular, p defines the probability of the environment entering to state $s' \in S_t$ and the agent receiving reward $r \in R_t$, after taking an action a in a state s at the previous step $t-1$. As a consequence, the probability of receiving reward R_t and entering state S_t depends only on S_{t-1} and A_{t-1}. Therefore, the state embodies all the information that can make a difference for the future. This property of states is called Markov property. From function p, other useful probability distributions can be obtained. In summary, the dynamics function describes all the useful aspects of the behaviour of the environment.

In MDPs, the level of abstraction of actions and states is not explicitly defined. Actions can be low-level controls or high-level instructions. At the same time, states can range from low-level sensory information to high-level and abstract observations. Time steps are generally defined as successive phases where a decision must be made. In summary, actions represent decisions to be learned, and states represent the information that can be used to make such decisions.

Solving RL tasks requires to specify the boundary between the agent and the environment. In some cases, many agents act simultaneously, and each agent operates within its own boundary. The environment is generally considered as anything that the agent cannot arbitrarily change. In particular, the reward computation must be external to the agent because it is defined as part of the task, and it should not be modified by the agent. In summary, the boundary between the agent and the environment is the limit of what the agent can completely govern. In MDPs, learning to solve a particular task requires three signals: action, state and reward. MDPs are simplified models of real

decision-making problems, but they have proved to be effective and convenient.

In some applications, it may be useful to build a model of the environment. The model simulates the environment, and it allows the agent to make predictions. In particular, a model is used to predict the next state s' and the obtained reward r, which are both the consequences of the agent taking an action a in a state s. A model can be very useful for planning, a RL algorithm that employs a model is called model-based. Otherwise, a RL method which learns a policy by simply trying possible actions without using a model is called model-free. Moreover, hybrid methods have been investigated. These approaches try to simultaneously learn both a model, which can be used to plan future actions, and the policy, by trial-and-error.

2.1.2 Reward

In order to define the goal of an agent for a particular task, a numerical reward signal $R_t \in \mathcal{R}$ must be defined. As mentioned before, a RL agent tries to maximize the cumulative reward in the long term. Rewards specify which actions are good or bad in a given state, in a similar way to the experiences of pleasure and pain that are observable in the animal world. For example, if the agent is a cleaning robot, a proper reward could be -1 for the agent when it collides with other objects. Otherwise, if the robot correctly cleans a portion of the room, the received reward could be $+1$.

The design of a reward system is not a trivial task because rewards must be given so that the agent achieves intended goals by maximizing them. It is not uncommon to give a reward when the agent reaches a sub-goal, and this choice typically results in the agent achieving sub-goal without reaching the goal. In summary, rewards must be shaped to tell the agent what to achieve, instead of communicating the sequence of actions needed to complete the task.

2.1.3 Episode

Saying that an RL agent tries to maximize the cumulative reward is imprecise. In fact, an agent tries to maximize the expected return. The return is a function of the rewards sequence, and it can be defined as the sum of rewards between two time steps. Formally, the expected return is defined as:

$$G_t = \sum_{k=0}^{T} R_{t+k+1} \tag{2.2}$$

This definition makes sense if the interaction between the agent and the environment can be naturally subdivided into a sequence of episodes. An episode is an independent sub-sequence of interactions that ends in a terminal state. An episode is always followed by a reset of the environment to the initial state or to a state chosen from a standard distribution of initial states. For example, in board games, like Go, an episode is typically a match, and the terminal state is reached at the end of a match. A task that can be structured in episodes is called an episodic task. Otherwise, if the interaction continues without reaching a terminal state, the task is a continuing task. In continuing tasks, the terminal state could be impossible to reach. This represents a problem with the definition 2.2 because if $T = \infty$, G_t cannot be computed easily. A solution to this problem is the concept of discounting. In particular, the agent tries to maximize the expected discounted return defined as:

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.3}$$

where $\gamma \in [0, 1]$ is the discount rate. Using definition 2.3, G_t has a finite value if $\gamma < 1$ or the reward sequence is bounded. The discount rate changes the learning process of the agent, which prefers immediate rewards if γ is close to 0. Otherwise, setting γ close to 1 makes the agent more farsighted, as it takes future rewards into account. As discussed earlier, considering long-term rewards tends to maximize the expected return.

An important relation between successive time steps can be obtained by imposing a null expected return after T time steps: $G_{t+1+T} = 0$. In this case, the expected discounted return is defined as:

$$G_t = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots \gamma^{t+k-2} R_{t+k}) = R_{t+1} + \gamma G_{t+1} \tag{2.4}$$

The above definition is crucial to the RL theory because it directly represents the return G_t as the sum of the immediate reward and γG_{t+1}, which is the expected discounted return at time step $t + 1$.

2.1.4 Policy

A policy describes the behaviour of the agent, which may be stochastic. A policy is typically represented as a mapping from an observed state to the action taken by the agent in that state. Stochastic agents require the policy to specify a probability for each action, given an observed state. A policy is a concept similar to the stimulus-response association that is theorized in Psychology.

In order to represent a policy, a function or a lookup table is sufficient. However, many real applications require complex and expensive functions to represent the policy of the agent. In general, a policy π maps states to actions probabilities, and it is defined as the following probability distribution:

$$\pi(a|s) = P\{A_t = a|S_t = s\} \tag{2.5}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. As the learning progresses, the policy is changed by the RL method, modifying the interaction between the agent and the environment, as a consequence. Rewards are used to change the policy, enforcing actions followed by high rewards, and discouraging actions followed by low rewards. Rewards are generally hard to predict as they are typically stochastic functions of state-action pairs.

2.1.5 Value function

While rewards represent the best immediate return, the value function is used to describe the expected return that the agent expects to obtain from a specific state. Rewards are similar to pleasure and pain, while values represent how much pleased the agent is expected to be in an given state. For example, the agent can be in a state which simultaneously results in a low reward and high value because the subsequent states often yield high rewards. The opposite case can be equally possible.

State values must be continuously re-estimated as the learning progresses, and many RL methods make use of a technique to efficiently estimate values. Formally, value functions $V_\pi(s)$ are defined with respect to a policy π and to a state $s \in \mathcal{S}$:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.6}$$

for all $s \in \mathcal{S}$, where $\mathbb{E}_\pi$ is the expected value of a random variable assuming that the agent makes use of policy π. The function V_π is also known as the state-value function for policy π. As a special case, the terminal state has value 0.

During the learning process, it is necessary to compare two policies and pick the best one. It is possible to define an ordering on policies using the following relation:

$$\pi \geq \pi' \quad \longleftrightarrow \quad \forall s \in \mathcal{S}, V_\pi(s) \geq V_{\pi'}(s) \tag{2.7}$$

Using the above definition, it is possible to define the optimal policy π_*:

$$\forall \pi \in \Pi, \quad \pi_* \geq \pi \tag{2.8}$$

where Π is the policy space. In order to measure the quality of a policy, the value function can be used.

An important function that is often used in place of V_π is the action-value function for policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.9}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. The action-value function represents the expected discounted return assuming that the agent starts from state s and it takes action a. Many RL methods do not directly use state-value or action-value functions. They use the advantage function, which is defined as:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{2.10}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. The advantage function denotes the convenience, in terms of return, for the agent to select action a with respect to the other actions, in a state s.

There are different class of RL methods, and some methods are directly based on value functions. In particular, an agent can use policy π maintaining an average of the returns for each encountered state. As the number of encountered states tends to infinity, the average value converges to $V_\pi(s)$. Similarly, the average of the obtained return that has been taken for each action-state pair converges to $Q_\pi(s,a)$.

Maintaining an average value of the returns is not practical in most real scenarios. Bellman equations are the fundamental relationships that allow a RL method to iteratively estimate V_π and Q_π. They describe the relationship between the value of a state and the values of its successors, and they are defined as:

$$\begin{aligned} V_\pi(s) &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)[r + \gamma V_\pi(s')] \\ Q_\pi(s,a) &= \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a',s')Q_\pi(s',a')] \end{aligned} \tag{2.11}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. The agent takes an action following policy π, and the environment enters the state s' and it outputs reward r. Bellman equations take into account all the possible execution paths, and they average over all the possibilities that are weighted by their probability. Using optimal policies, it is possible to define the optimal value functions V_* and Q_* as:

$$\begin{aligned} V_*(s) &= \max_\pi V_\pi(s) \\ Q_*(s,a) &= \max_\pi Q_\pi(s,a) \end{aligned} \tag{2.12}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. Functions V_* and Q_* are called optimal state-value function and optimal action-value function, respectively. Optimal value functions must fullfill the

conditions given by Bellman equations as well. Therefore, it is possible to define Bellman equations for optimal value functions:

$$\begin{aligned} V_*(s) &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V_*(s')] \\ Q_*(s,a) &= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} Q_*(s',a')] \end{aligned} \tag{2.13}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. The above equations are called *Bellman optimality equations*. A theoretical solution to RL problems is to find optimal value functions because a greedy policy with respect to the optimal value functions is an optimal policy. However, it is rarely possible to explicitly solve Bellman optimality equations. In fact, this approach requires that:

- The agent knows the dynamics of the environment; and
- The computational resources are sufficient to complete the search for the solution.

These assumptions are rarely true in real problems. Therefore, RL methods try to find approximate solutions because, in many cases, taking sub-optimal actions does not have a great influence on the amount of obtained reward in the long run. Learning to make good choices in frequently encountered states, and paying less attention to unfrequently encountered states, is a key characteristic of RL with respect to other approaches that approximately solve MDPs.

2.1.6 Applications

RL can address many problems. In particular, games are often used to test the performance of RL methods. Many perfect information games have been studied, such as Backgammon [25] and Go [26]. Imperfect information games have been explored as well, such as Heads-up Limit Hold'em Poker [5]. In recent years, video games have been increasingly investigated. For example, the Atari 2600 games [27] are some of the most relevant benchmark tasks for RL methods, but many improvements have been made on Starcraft [28], Doom [29] and many other games.

Other important applications of RL are robotics and *Natural Language Processing* (*NLP*). Popular robotics tasks include object manipulation, localization, navigation, and visual tracking. Common NLP tasks include information retrieval, summarization, and sentiment analysis. It is worth noting the relevant research effort made in NLP areas such as machine translation, dialogue systems, and text generation.

Finally, RL represents a valid approach to solve computer vision tasks, and it influences many other research fields. Examples of computer vision tasks, which are commonly

addressed by RL methods, are object segmentation, object recognition or categorization, and grasp planning.

2.2 RL techniques

This section overviews the main approaches to RL: *Dynamic Programming* (*DP*), *Monte Carlo* (*MC*), *Temporal-Difference* (*TD*), and *Policy Gradient* (*PG*). The approaches are briefly presented and compared in order to give the reader an overview of the classic RL literature, which represents the foundation of modern RL methods.

2.2.1 Dynamic Programming

Examples of model-based RL algorithms are DP algorithms [1]. DP methods require a perfect model of the environment, as well as a large amount of computational resources. Therefore, they are not widely used, although they are theoretically capable of finding an optimal policy. The theory behind DP algorithms remain useful, and it can be used to solve problems with continuous spaces of actions and states. However, obtaining exact solutions is possible only in special situations, and these methods typically search for approximate solutions. In general, approximate solutions are computed by discretizing the spaces of actions and states, and then applying finite-state DP algorithms.

Let assume that the dynamics function p is known, then it is possible to use Bellman equations to iteratively approximate the state-value function V_π for a policy π. This problem is called policy evaluation. DP methods define the initial approximation V_0, which outputs an arbitrary value for any states. The only exception is the terminal state, which must have a value of 0. In order to compute the solution, the algorithm employs the following update rule:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')] \tag{2.14}$$

for all $s \in \mathcal{S}$. The update rule is iteratively applied until the approximation reaches a fixed point. State-value function V_π is guaranteed to exist and to be unique if the following conditions are met:

- The discount rate is opportunely bounded, i.e., $\gamma < 1$; and
- The termination is ensured starting from all states and following policy π.

Moreover, if the conditions are met, the procedure is guaranteed to converge as k tends to infinity. Assuming that V_π has been determined using policy evaluation for a deter-

ministic policy π, let π' be another deterministic policy such that:

$$\forall s \in \mathcal{S} \quad Q_\pi(s, \pi'(s)) \geq V_\pi(s) \tag{2.15}$$

In this case, π' is equal or better than π. In particular, the above inequality is useful to find possible improvements that are determined by a modification of the current policy π. As a consequence, the following relation holds:

$$V_{\pi'}(s) \geq V_\pi(s) \tag{2.16}$$

for all $s \in \mathcal{S}$. It is worth noting that if the inequality 2.15 is strict, then the above inequality is strict as well. The above inequality allows to define a new policy π' from an existing policy. This process is called policy improvement. In order to obtain π' the update rule is defined as:

$$\pi'(s) = \operatorname*{argmax}_a Q_\pi(s, a) = \operatorname*{argmax}_a \sum_{s', r} p(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.17}$$

for all $s \in \mathcal{S}$. Policy improvement is useful to iteratively build an optimal policy, either deterministic or stochastic.

Combining policy evaluation and policy improvement allows obtaining both an optimal policy and an optimal value function in a finite number of iterations, when the problem is modeled using a finite MDP. In particular, policy evaluation is used to obtain a value function V_{π_0} from an initial policy π_0. Then, policy improvement is used to obtain a new policy π_1. The entire process is called policy iteration, and it can be represented as follows:

$$\pi_0 \xrightarrow{\text{E}} V_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} V_*$$

where $\xrightarrow{\text{E}}$ denotes policy evaluation, and $\xrightarrow{\text{I}}$ indicates policy improvement.

Policy iteration can be generalized defining two independent processes. The first process implements policy evaluation using the current policy π. The second process implements policy improvement using the current value function V_π. This idea is called *Generalized Policy Iteration* (*GPI*), and it has been proved to converge only in some cases, as discussed in [1].

DP methods re-estimate the values of the states using the values of their successors. The idea of updating an approximation using another approximation is called bootstrapping, and it is shared also by TD learning, which is discussed in 2.2.3.

2.2.2 Monte Carlo

MC methods [1] are examples of model-free RL algorithms, and they assume that the interaction between the agent and the environment is subdivided into episodes. Each episode must terminate because these methods update the policy only after the termination of the episode. In particular, MC algorithms compute the value of a state s as the average return of all sample trajectories starting from s. MC methods are composed of policy evaluation and policy improvement, in a similar way to DP.

MC methods collect complete episodes, and they try to empirically learn the value function. Formally, the state-value function V_π for MC algorithms is defined as:

$$V_\pi(s) = \mathbb{E}[G_t|S_t = s] = \frac{1}{N}\sum_{i=1}^{N} G^i_{t,s} \tag{2.18}$$

for all $s \in \mathcal{S}$. As discussed in previous sections, the expected discounted reward at time step t is defined as $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$. Here, $G^i_{t,s}$ denotes the expected discounted reward considering state s, episode i, and N as the number of episodes. V_π can be computed by averaging returns $G^i_{t,s}$ every time the agent encounters state s or the first time the state is visited. In a similar way to V_π, the action-value function can be defined as:

$$Q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] = \frac{1}{N}\sum_{i=1}^{N} G^i_{t,s,a} \tag{2.19}$$

for all $s \in \mathcal{S}$, and $a \in \mathcal{A}(s)$. $G^i_{t,s,a}$ denotes the expected discounted reward considering state s, and taking action a during episode i.

Computing the average return using the entire episode set is typically not convenient. Therefore, MC algorithms normally perform incremental updates to progressively learn as the number of studied episodes increases. The procedure to learn an optimal policy for MC methods is similar to policy iteration, and it can be summarized as:

- The policy π is updated using a greedy approach with respect to the current action-value function: $\pi(s) = \operatorname*{argmax}_a Q(s,a)$;
- A new episode is generated using policy π; and
- The value function Q is re-estimated using the new episode.

Unfortunately, naïve MC methods tend to exploit too much the learned experience. In order to solve this problem, exploration can be encouraged using two techniques: exploring starts and ϵ-soft. Exploring starts requires that all state-action pairs have non-zero probability of being chosen as initial state and action. Therefore, every time the

agent plays a new episode, it should explore new states. Many environments do not allow multiple starting states, in this cases ϵ-soft can be used. ϵ-soft requires the agent to try the action that maximizes the function Q, with probability $1 - \epsilon$, and to try a random action with probability ϵ. Following this strategy, all actions are tried with non-zero probability, if $\epsilon > 0$.

An important classification in RL is whether a methods is on-policy or off-policy. On-policy algorithms follow the policy that is currently updated. Off-policy algorithms try to learn an optimal policy that may be unrelated to the policy that is used to take actions. In particular, off-policy methods try to learn the value function of a target policy using the episodes that are generated by another policy, which is called behaviour policy.

The main differences between MC and DP methods are two. The first difference is that MC algorithms make use of sample experiences. Therefore, they do not require a model to learn a policy. The second difference is that MC methods do not employ bootstrapping, which updates value estimates using other values estimates.

2.2.3 Temporal-Difference

A recent RL approach is TD learning [1], which combines some ideas from MC and DP. TD learning is a model-free approach like MC but TD methods employ bootstrapping, in a similar way to DP methods.

TD learning employs a form of GPI, and the policy evaluation phase, also called TD prediction, is similar to the one that was discussed for MC methods. In particular, given a non-terminal state S_t at time step t, TD methods collect episodes following policy π, and they update an estimate of V_π accordingly. MC methods require to complete the current episode to compute the update of $V(S_t)$, where $V(S_t)$ represents the value estimate of state S_t. Therefore, an update rule for MC methods, assuming a non-stationary environment, can be written as:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{2.20}$$

where G_t is the obtained actual return starting from time step t, and α is a constant step-size parameter. Unlike MC methods, TD algorithms update the value estimate at each time step. In particular, they make use of reward R_{t+1} and the value estimate of the next state $V(S_{t+1})$. The update rule for TD methods can be written as:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.21}$$

The update rules of both approaches make use of different target values to optimize the value function. MC employs G_t, while TD uses $R_{t+1} + \gamma V(S_{t+1})$. This form of TD

learning is called $TD(0)$, or one-step TD. More advanced forms of TD algorithms exist in the literature, such as $TD(\gamma)$ and n-step TD. The update rule 2.21 is an example of bootstrapping: the value estimate $V(S_t)$ is updated using another value estimate $V(S_{t+1})$. The difference between the target value and the actual value is called TD-error, and it represents a recurrent concept in the RL literature. Formally, TD-error is defined as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.22}$$

The interesting characteristics of TD algorithms, such as learning without a model and updating the value estimates after each time step, make these methods appealing for a wide range of applications. From a theoretical point of view, TD(0) has been proven to converge to V_π for any policy π by accurately adjusting hyper-parameter α. However, it has not been proved that TD algorithms converge faster than MC methods.

TD prediction is the counterpart of policy evaluation for MC methods. TD learning follows GPI, and therefore it defines a counterpart for policy improvement, which is called TD control. Here, the two main approaches for TD control are presented: Sarsa and Q-learning. In particular, Q-learning is the base of many modern RL methods [12, 30–32].

Sarsa

Sarsa is an on-policy algorithm, and it estimates the action-value function of the policy that is currently used to generate episodes. In particular, Sarsa defines the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.23}$$

which is very similar to the update rule used by TD prediction. Using the above rule, Sarsa continually re-estimates Q_π for each state s and action a. Then, it changes the policy π to be greedy with respect to Q_π. The action-value function is updated after each transition from a non-terminal state S_t to a state S_{t+1}. The value $Q(S_{t+1}, A_{t+1})$ is defined as 0 if S_{t+1} is a terminal state.

Sarsa can be summarized as follows:

1. The agent takes action A_t, assuming state S_t at time step t. In order to select the action, the agent uses Q, commonly applying an ϵ-soft strategy;

2. The agent receives reward R_{t+1} and it enters state S_{t+1};

3. The agent takes action A_{t+1} from state S_{t+1} using the strategy in 1;

4. The agent updates $Q(S_t, A_t)$ using the update rule 2.23;

5. The agent increases the time step number, and it repeats from 1.

Sarsa converges to an optimal policy and an optimal action-value function if all state-action pairs are encountered an infinite number of times. Moreover, how much the policy depends on Q determines the convergence of Sarsa. In particular, ϵ-soft can be used to reduce such dependency, and Sarsa converges to an optimal policy and an optimal Q function as long as the policy converges to the greedy policy as t tends to infinity.

Q-learning

Q-learning represents an important result for RL theory. The update rule for Q function approximation is defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.24}$$

Q-learning tries to directly estimate the optimal action-value function Q_*. The policy is not directly approximated but Q-learning assumes that the greedy policy with respect to Q is used. The algorithm can be summarized as follows:

1. The agent takes action A_t, assuming state S_t at time step t. Action A_t is selected accordingly to Q, and strategies such as ϵ-soft are commonly used;

2. The agent receives reward R_{t+1} and it enters state S_{t+1};

3. The agent updates $Q(S_t, A_t)$ using the update rule 2.24;

4. The agent increases the time step number, and it repeats from 1.

The steps 1 and 2 are shared with Sarsa. Step 3 is used to estimate Q_* taking the best Q values without using the current policy. It is worth noting that the policy is still used to decide which state-action values are updated. The convergence of Q-learning is guaranteed as long as all state-action pairs are visited and continuously updated. Moreover, specific conditions on the sequence of step-size parameters must be met to ensure the convergence of Q-learning [1].

2.2.4 Policy Gradient

PG methods [33] do not try to learn a value function to perform action selection, like the previously discussed approaches. PG algorithms try to directly learn the policy using a function parameterized with respect to θ: $\pi(a|s, \theta)$. Let J be a reward function

representing the expected return that the agent obtains using policy π. In particular, J is defined for discrete state and action spaces as:

$$J(\theta) = V_{\pi_\theta}(S_1) \tag{2.25}$$

where θ are the parameters that define the behaviour of the policy, and S_1 is the initial state. J is defined using θ, and the agent must optimize θ to increase the obtained rewards. The reward function for continuous action and state spaces is defined as:

$$J(\theta) = \sum_{s \in S} d_{\pi_\theta}(s) V_{\pi_\theta}(s) = \sum_{s \in S} (d_{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s,\theta) Q_\pi(s,a)) \tag{2.26}$$

where $d_{\pi_\theta} = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta)$ denotes the probability of reaching state s starting from the initial state, and following policy π_θ. PG methods are commonly employed to solve continuous space problems. In fact, in continuous space problems, the number of states or actions is infinite, and estimating value functions requires a huge amount of computational resources.

In order to optimize a set of parameters, a commonly employed technique is gradient ascent. Gradient ascent iteratively adjusts θ following the direction that is given by gradient $\nabla_\theta J(\theta)$. In particular, gradient ascent computes the partial derivatives of $J(\theta)$ with respect to each parameter in θ. Computing gradients with respect to $J(\theta)$ is useful to move θ in the direction that maximizes the expected return. Therefore, θ is moved towards the direction suggested by $\nabla_\theta J(\theta)$ to find optimal parameters θ_*, which define the optimal policy π_{θ_*}.

Gradient $\nabla_\theta J(\theta)$ is difficult to obtain because it relies on both action selection and states distribution d_{π_θ}. Both action selection and states distribution depend on π_θ, and it is typically difficult to predict the consequences of a policy update when the environment is unknown. An important theoretical result that allows to estimate the reward function is the Policy Gradient Theorem [33]. Using Policy Gradient Theorem, the expected return $J(\theta)$ can be computed as:

$$J(\theta) \propto \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s,\theta) Q_{\pi_\theta}(s,a)] \tag{2.27}$$

Policy Gradient Theorem represents a theoretical foundation for many PG algorithms, and it can be used to approximate reward function J without knowing the state distribution d_{π_θ}. Unfortunately, the above relation produces policy updates with no bias and high variance, and a significant research effort has been done to reduce the variance without changing the bias. An interesting algorithm that were introduced before PG methods is REINFORCE.

REINFORCE

REINFORCE [34] combines PG and MC, and it uses episode samples to compute the estimated return. The algorithm employs the estimated return to update θ. In particular, REINFORCE estimates the reward function J using the following relation:

$$J(\theta) \propto \mathbb{E}_{\pi_\theta}[\nabla_\theta \ln \pi(a|s,\theta) G_t] \tag{2.28}$$

Basically, it rewrites the relation 2.27 replacing $Q_{\pi_\theta}(s,a)$ with G_t. This is justified by $Q_{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}[G_t|S_t, A_t]$. REINFORCE is similar to MC algorithms because it uses full samples trajectories G_t to update the policy.

A common issue with both PG and REINFORCE algorithms is the high variance of policy updates, which can make the training unstable. In order to reduce the variance of the updates without changing the bias, the return G_t can be replaced in definition 2.28 by the following advantage function:

$$A_{\pi_\theta}(s,a) = G_t - V_{\pi_\theta}(s) \tag{2.29}$$

The above definition of the advantage function is different from the canonical one: $A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s)$, and G_t is used as an estimate of $Q_{\pi_\theta}(s,a)$. The modified advantage function 2.29 makes the training more stable and this variant of REINFORCE [1] is able to learn faster than the base algorithm.

Chapter 3

Deep and Relational Reinforcement Learning

Deep reinforcement learning (*DRL*) tries to combine classical *Reinforcement Learning* (*RL*) theory with modern deep learning techniques. In recent years, DRL has received an increasing interest from the research community, and many DRL algorithms obtained impressive results on complex tasks [3–6, 12]. On the other hand, *Relational Reinforcement Learning* (*RRL*) combines classic RL with a relational representation of both states and actions. RRL has been studied for more than 20 years, and in recent years, the interest in RRL has increased. In fact, the combination of RRL with modern deep learning techniques seems to be an effective way to overcome the limitations of classic RRL methods. An example of this combination is [9].

This chapter presents some of the most representative results on DRL related to the last few years. In addition, this chapter introduces the classic RRL algorithms, and it presents the main differences between DRL and RRL. Moreover, this chapter analyzes the main limitations of both approaches, and it motivates the need of neural-symbolic methods for RL. The chapter is structured as follows. Section 1 introduces the most influential DRL methods. In particular, Section 1 presents DQN [3, 12], which combines deep learning with Q-learning. Then, it introduces A3C [13], which combines deep learning with a Policy Gradient algorithm. Then, it discusses ACER [14], which is the off-policy counterpart of A3C. Finally, it briefly presents PPO [6], which is one of the most recent breakthroughs that have been introduced in the field. Section 2 introduces the research field of RRL. Then, it presents the classic RRL method [7], which adapts Q-learning to work with relational representations of both the states and the actions. Finally, it briefly introduces three notable extensions of the classic RRL method [35–37]

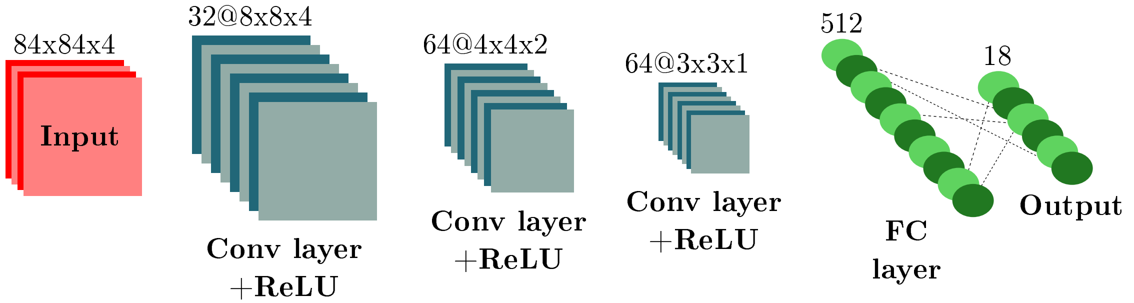

Figure 3.1: The neural network architecture of DQN.

that have been proposed in the first years of research on RRL. Section 3 concludes this chapter discussing the limitations of both DRL and RRL as well as the motivation behind the neural-symbolic methods for RL.

3.1 Deep RL

3.1.1 DQN

Q-learning requires to memorize the action value $Q(s, a)$ for each state $s \in \mathcal{S}$ and each action $a \in \mathcal{A}(s)$. In realistic applications, this is often impossible to achieve because both the state space and the action space are very large. Therefore, the action-value function Q is typically used as a function approximator. *Deep Q-learning* (*DQN*) [3, 12] uses a deep neural network as the function approximator for Q. If θ denotes the parameters of the neural network, the approximated action-value function can be written as $Q_\theta(s, a)$.

Combining Q-learning with a deep neural network is not a trivial task because the training process is not guaranteed to converge, and it could also suffer from instability. In order to reduce the convergence and stability problems, DQN proposes two important innovations:

- **Experience Replay**
  The collected samples are stored in a special memory called replay memory, which can contain a very large number of elements. DQN randomly selects a batch of samples from the replay memory during each Q-learning update. Therefore, each sample can be used multiple times. Note that experience replay has not been introduced with DQN [38]. However, DQN combines experience replay with Q-learning to improve data efficiency, and to remove the correlations between subsequent observations, that is normally present in Q-learning when used to learn how to play at Atari 2600 console games. As a consequence, experience replay reduces the variance of the Q-learning updates, smoothing over changes in the samples distribution.

- **Periodically updated target**

  Q-learning requires to frequently change the target value when computing TD-error. In order to stabilize the training process and to avoid unnecessary oscillations, Q-values are optimized using a target value that is periodically updated. In particular, the parameters of the neural network are updated every C time steps, where C is an hyper-parameter.

One of the most impressive achievements of DQN is that the algorithm can be used to learn how to play the games of Atari 2600 consoles using frames that are provided by an emulator [27]. Learning from realistic images (210x160 pixels with a color palette of 128) can be computationally expensive. Therefore, DQN computes Q-values updates every m frames to reduce the frequency of actions to be taken (one action is executed for m consecutive frames) and the number of Q-values updates. Moreover, DQN performs many other optimizations by changing the representation of the processed frames.

Rewards are represented by a wide range of values, and those values depend on specific events. When the rewards have high variance, the training process can become unstable. In order to improve the training stability, DQN clips the rewards, and it sets positive rewards to $+1$ and negative rewards to -1.

The network architecture of DQN is composed of an input layer, followed by three convolutional layers that are useful to extract high-level features from input images. The convolutional layers are followed by a fully-connected layer and an output layer, which reports the predicted Q-value for each possible action. The architecture is represented in Figure 3.1. DQN proved to be effective because its architecture is able to compute Q-values of a given state for each action in a single pass through the network. Interestingly, many extensions of DQN have been proposed, such as Double DQN [32], Dueling DQN [30], and Prioritized DQN [31].

3.1.2 A3C

Asynchronous Advantage Actor-Critic (*A3C*) [13] is a Policy Gradient algorithm that follows the actor-critic paradigm. Actor-critic methods try to learn both policy and value function. In particular, they try to learn the value function that is used to reduce the variance of policy updates. Actor-critic algorithms define two components, whose parameters my be shared:

- **Critic**

  The critic tries to learn a value function, either $Q_w(s, a)$ or $V_w(s)$, where w represents the parameters that the value function uses. The critic updates w to learn

the value function.

- **Actor**
  The actor tries to learn policy $\pi_\theta(a|s)$ with parameters θ, which are updated in the direction that is indicated by the critic.

A3C defines multiple actors, which try to learn both the value function and the policy. The actors are executed in parallel and the global parameters are periodically synchronized. Assuming that the actors try to learn the state-value function, the loss is defined as $J_v(w) = (G_t - V_w(s))^2$, and gradient ascent is used to find the best values for weights w. Gradient ascent computes the partial derivatives of the value function with respect to the parameters of the network, also called gradients. The network parameters are then updated in the direction that is specified by the gradient. The entire process tries to iteratively move the values of the parameters in the direction that maximize the value function. A3C uses the values function as a baseline, and it accumulates gradients with respect to w and θ. Each training thread gives its contribution to the parameters update. This training procedure can be seen as a parallelized reformulation of the gradient update based on minibatches because updates are accumulated over multiple time steps before being applied to the weights.

Both the value function and the policy are approximated using a deep neural network. The network architecture is very similar to the one used by DQN. However, A3C defines a neural architecture with two output layers. The first output layer implements a softmax policy $\pi(a|s)$. The second output layer computes the value of the current estimate of either $V(s)$ or $Q(s, a)$, depending on the value function that is used for training.

3.1.3 ACER

Actor-Critic with Experience Replay (*ACER*) [14] is the off-policy counterpart of A3C. ACER uses experience replay, and it employs the same architecture of DQN with two output layers: the first computes $\pi_\theta(a|s)$, and the second computes $Q_{\theta_v}(s, a)$. Similarly to A3C, ACER collects samples using multiple threads. It is not a complete off-policy algorithm: it switches between on-policy and off-policy executions. In particular, an on-policy call (which works like A3C) is alternated to multiple off-policy calls. The number of off-policy calls is called the *replay ratio*. Using a replay ratio of 4, ACER performs in a similar way to Prioritized DQN or A3C, but ACER is more sample efficient. Notably, ACER performs well on a wide range of RL problems, and it proved effective to solve even hard exploration problems [39].

ACER tries to increase sample efficiency and to decrease the correlation between selected samples, and it proposes three major innovations to stabilize the off-policy estimator:

- Retrace Q-value estimation;
- Importance weights truncation with bias correction; and
- Efficient Trust Region Policy Optimization.

Retrace

Retrace [40] is an off-policy TD learning algorithm. Similarly to DQN, Retrace uses experience replay to improve sample efficiency, and it typically explores more because it follows a policy that is different from the target policy. However, Retrace uses multi-step value estimation, which helps reducing the bias introduced by bootstrapping. TD-error for Retrace is defined as:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - Q(S_t, A_t). \tag{3.1}$$

Q-values are updated using the rule $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\delta_t$, which can be written also as $\Delta Q(S_t, A_t) = \alpha\delta_t$. TD-error δ_t is used to estimate Q_{π_θ} using a complete sample trajectory. However, following a policy different from the one that is updated, it introduces a bias. In order to remove this bias, off-policy algorithms typically use Imporance Sampling. The update rule with Importance Sampling becomes:

$$\Delta Q(S_t, A_t) = \gamma^t \left(\prod_{1 \leq \tau \leq t} \frac{\pi(A_\tau|S_\tau)}{\mu(A_\tau|S_\tau)} \right) \delta_t \tag{3.2}$$

where $\pi(a|s)$ and $\mu(a|s)$ are the target and behaviour policies, respectively. This update rule produces updates with unbounded variance. Retrace proposes a further modification to the rule 3.2 to solve the problem:

$$\Delta Q(S_t, A_t) = \gamma^t \left(\prod_{1 \leq \tau \leq t} \lambda \min \left\{ 1, \frac{\pi(A_\tau|S_\tau)}{\mu(A_\tau|S_\tau)} \right\} \right) \delta_t \tag{3.3}$$

The above update rule produces updates with bounded variance, and it assures convergence for any policy pair π, μ. ACER generates a sample trajectory using policy μ, and it uses Retrace to estimate $Q_{\pi_\theta}(S_t, A_t)$ using the following rule:

$$Q^{ret}(S_t, A_t) = R_{t+1} + \gamma \min \left\{ c, \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \right\} \left[Q^{ret}(S_t, A_t) - Q_{\theta_v}(S_t, A_t) \right] + \gamma V(S_{t+1}) \tag{3.4}$$

where Q_{θ_v} is the current approximation of Q_{π_θ}. Finally, ACER uses Q^{ret} to define the following loss function:

$$J_v(\theta_v) = (Q^{ret} - Q_{\theta_v})^2 \tag{3.5}$$

where Q^{ret} is used as the target function to update Q_{θ_v}.

Importance weight truncation

Importance sampling is useful to reduce the variance of the Q-values updates. However, using importance sampling introduces a bias, and ACER uses a correction term to solve this problem. Therefore, the gradient of the policy at time step t is written as:

$$\begin{aligned}\hat{g}^{acer} &= \bar{\rho}_t(Q^{ret}(S_t, A_t) - V_{\theta_v}(S_t))\nabla_\theta \ln \pi_\theta(A_t|S_t) \\ &+ \mathbb{E}_{a\sim\pi}\left[\max\left\{0, \frac{\rho_t(a) - c}{\rho_t(a)}\right\}\nabla_\theta \log \pi_\theta(a|S_t)(Q_{\theta_v}(S_t, a) - V_{\theta_v}(S_t))\right]\end{aligned} \tag{3.6}$$

where $\bar{\rho}_t = \min\left\{c, \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}\right\}$. The first term of the above equation represents the bounded importance weight, and the second term represents the correction term that is used to simultaneously remove the bias and reduce the variance of the update.

Efficient Trust Region Policy Optimization

As previously mentioned, off-policy methods define the gradient of the policy using Importance Sampling. In particular, the gradient for off-policy algorithms is written as:

$$\nabla_\theta J(\theta) = \mathbb{E}_\mu\left[\frac{\pi_\theta(a|s)}{\mu_\theta(a|s)} Q_{\pi_\theta}(s, a)\nabla_\theta \ln \pi_\theta(a|s)\right] \tag{3.7}$$

where $\frac{\pi_\theta(a|s)}{\mu_\theta(a|s)}$ is called *importance weight*.

Trust Region Policy Optimization (*TRPO*) [41] is used to improve the stability of the training process, and it can be applied either to on-policy or off-policy algorithms. TRPO enforces a constraint (*trust region constraint*) on $J(\theta)$. In particular, the distance between old and new policies may not exceed the value $\delta \in \mathbb{R}$. Therefore, the updates of parameters do not change too much the policy in a single step. The distance between two policies is measured using *Kullback-Leibner* (*KL*) divergence [42], which measures how much a probability distribution q diverges from another probability distribution p, and it is denoted as $D_{\mathrm{KL}}(q||p)$. The objective function $J(\theta)$ is rewritten as:

$$J(\theta) = \mathbb{E}_{s\sim\rho_{\pi_{\theta_{\mathrm{old}}}}, a\sim\pi_{\theta_{\mathrm{old}}}}\left[\frac{\pi_\theta(a|s)}{\theta_{\mathrm{old}}(a|s)} A_{\theta_{\mathrm{old}}}(s, a)\right] \tag{3.8}$$

where $\pi_{\theta_{\mathrm{old}}}$ are the parameters of the policy before the update, and $\rho_{\pi_{\theta_{\mathrm{old}}}}$ is the state visit distribution of the old policy. The constraint that TRPO enforces can be written

as:

$$\mathbb{E}_{s\sim\rho_{\pi_{\theta_{\text{old}}}}}[D_{\text{KL}}(\pi_{\theta_{\text{old}}}(\cdot,s)\|\pi_\theta(\cdot,s))] \leq \delta \tag{3.9}$$

TRPO measures the KL divergence between an initial policy and an updated policy. ACER employs a modified version of TRPO. In particular, it maintains a running average of previous policies, and it changes the policy update to prevent the policy deviating too much from the average. The modified TRPO algorithm is more computationally efficient, and it stabilizes the learning process.

3.1.4 PPO

Proximal Policy Optimization (*PPO*) [6] is a PG algorithm that tries to combine the efficiency and the reliability of TRPO in a simplified way. In fact, TRPO is quite complicated, and it is not always usable. PPO proposes a new objective function $J(\theta)$, which is somewhat similar to the one that has been proposed by TRPO:

$$J(\theta) = \mathbb{E}_{s\sim\rho_{\pi_{\theta_{\text{old}}}},a\sim\pi_{\theta_{\text{old}}}}[\min(r(\theta)A_{\theta_{\text{old}}}(s,a), clip(r(\theta)A_{\theta_{\text{old}}}(s,a), 1-\epsilon, 1+\epsilon)A_{\theta_{\text{old}}}(s,a))]$$

where $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$, and ϵ is an hyper-parameter. The first term inside the min operator is the same objective used by TRPO 3.8, which is denoted here as L^{TRPO}. PPO clips $r(\theta)$ to prevent the probability ratio from going outside the interval $[1-\epsilon, 1+\epsilon]$. Then, the algorithm takes the minimum between the unbounded L^{TRPO} and the clipped L^{TRPO} to obtain a lower bound on the unclipped objective. In particular, when $A_{\theta_{\text{old}}} > 0$ and $r(\theta) < 1-\epsilon$, a large update of the policy has been made, which decreases the probability of taking better actions. At the same time, when $A_{\theta_{\text{old}}} < 0$ and $r(\theta) > 1+\epsilon$, a large policy update has been made, which increases the probability of taking worse actions. In both cases, the algorithm uses the min operator to pick the unbounded L^{TRPO}, rectifying an erroneous update with another large update that goes in the opposite direction.

PPO has been tested on a large number of tasks, and it performed very well. In particular, PPO shows performance that are similar to the ones reported by ACER [14]. However, it is less complex, and it can be easily implemented. Moreover, it is more flexible because it removes the trust region constraint, which prevents TRPO from being used with neural architectures that include noise or parameters sharing.

3.2 Relational RL

DRL methods produce models that are not interpretable. Moreover, these algorithms are not able to generalize to tasks that are different from the one used for training. In

particular, the majority of real-world problems can be naturally represented using a relational representation, in which the environment is represented by a set of objects and their relationships. For example, states, actions, and goals could be represented using first order logic. Learning from a relational representation of the task has several advantages. For example, the agent can learn a strategy that solves the task of manipulating a set of objects, but the learned strategy can be used to solve the same task using a set of different objects without being retrained. Moreover, the agent can tackle different tasks that share a similar representation, allowing the transfer of knowledge across tasks. In particular, the agent can be trained on a simple task, and the learned strategy can be used to improve the learning performance on more complex tasks. Finally, it is worth noting that a relational representation of a task allows using an extensive and appropriate background knowledge. Therefore, domain experts can reuse their valuable and domain-specific knowledge to increase the performance of the method as well as the quality of the learned strategy.

RRL [7] is a form of RL in which the agent learns from a relational representation of both states and actions. Combining a structured background knowledge with the relational representation of states and actions, RRL reduces the number of different situations that need to be handled. Therefore, RRL is able to reuse the learned knowledge to effectively solve different problems with a single training process. The first influential paper in RRL is the classic RRL method, proposed by Džeroski in 1998 [7]. The classic RRL method is an adaptation of Q-learning, where the action-value function Q is learned using a first-order logic regression tree algorithm [43]. A logic regression tree is a tree in which the leave nodes contain a value that represents a prediction for a continuous class, and the internal nodes contain tests that partition the example space. The example are a set of facts representing the state, the action to be taken, and the goal. The tests that are represented in the internal nodes contain predicates, variables and complex terms. The classic RRL method builds a first-order logic regression tree (or relational regression tree) that takes as input the state representation (a set of ground atoms), a goal, and an action, and it returns the associated Q-value. In particular, the algorithm uses Q-learning to continuously update the current estimate of the Q function. The original Q-learning algorithm updates the current estimate of the action-value function for the observed state-action pair as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \tag{3.10}$$

The classical RRL method generates a tuple (S_t, A_t, Q_t) at each time step, and it employs the TILDE-RT algorithm [43] to produce the next estimate of the Q-value function. The

classic RRL method starts with an initial relational tree $\hat{Q}_0$ that maps each state-action pair to the value 0. For each goal state g that is defined, and for each action a that leads to g, the method generates the example $(g, a, 0)$. As a consequence, the algorithm does not expect any reward by taking an action in a goal state. The algorithm does not incrementally update the relational regression tree, but it generates a new tree at the end of each episode e. The generated $\hat{Q}_e$ is then used to take actions in the next episode. The classic RRL method keeps the most recent Q-value for each state-action pair, and it employs TILDE-RT to produce a relational regression tree. The internal nodes of the generated trees contain conditions that subdivide the example space, and these conditions are build using logical entities such as predicates, variables, and complex terms. The leaf nodes of the generated trees represent the Q-values associated to the state-action pairs. In summary, a relational regression tree is implemented by a Prolog program that returns a different Q-value depending on the set of atoms that defines the state, the action, and the goal. For example, the following Prolog program represents a logical regression tree for the ON task, as shown in [7]:

```
qvalue(0)   :- action(move(A,B)), goal(on(C,D)), on(C,D), !
qvalue(1)   :- action(move(A,B)), goal(on(C,D)), action(move(C,D)), !
qvalue(0.9) :- action(move(A,B)), goal(on(C,D)), action(move(D,B)), !
qvalue(0.81).
```

Here, the goal of the agent, described using predicate `goal`, is to put a specific block onto another specific block (represented by variables C and D, respectively). The agent can use the `move` action to move a block onto another block, and the environment is described using the `on(X,Y)` predicate, which expresses that a block `X` is onto another block `Y`. When running the query `?-qvalue(Q)`, the program returns the Q-value of the current state-action pair, provided that the state, the goal, and the action are described before the the execution of the query.

Different extensions of the classic RRL method have been proposed. The first notable extension has been proposed in [35], which combines the original algorithm with the TG algorithm to incrementally build the relational regression tree. The TG algorithm combines the TILDE algorithm [44] with the G algorithm [45]. The TG algorithm produces a tree that includes conjunctions of first order literals as the conditions in the internal nodes. Each leaf node includes a conjunction of the tests that are present in the path from the root node to the leaf node, and the variables included in the tests are existentially quantified. The algorithm stores different statistics in the leaf nodes, such as the number of positively classified examples as well as the number of negatively

classified examples. For each classified example, TG also stores the sum of the Q-values and the sum of the squared Q-values. These statistics are used to extend the tree, and to decide which tests should be inserted in the new nodes. Another notable extension of the classic RRL method is based on the RIB algorithm [36]. The RIB algorithm computes a weighted average of the Q-values of the samples that Q-learning produces. The algorithm computes the distances among the samples, which must take into account the relational representation of states and actions. The selected weights are inversely proportional to the distances among the samples. A third notable extension of the classic RRL method makes use of the KBR algorithm [37]. The KBR algorithm uses Gaussian processes [46] to learn the action-value function. A notable feature of KBR is that the algorithm can also indicate the expected accuracy of the predicted Q-value of a new sample. This information can be useful to guide the exploration during the training phase.

3.3 Discussion

Some of the most influential works on DRL have been discussed in this chapter. These techniques employ deep neural networks as function approximators. Unfortunately, artificial neural networks are very difficult to interpret, and the learned knowledge cannot be extracted and effectively reused. Moreover, many experiments show that neural networks are not able to generalize to tasks that are slightly different from the tasks used for training [47–49]. In fact, the discussed works report impressive performance on the chosen tasks but agents cannot tackle new tasks without specific training. Therefore, it is very difficult to transfer knowledge. For example, the agent cannot easily learn a strategy for a complex task starting from another strategy that has been learned from a easier version of the same task. Finally, it is not possible to easily exploit an existing background knowledge. In fact, a background knowledge can improve the performance of RL method because the agent is not forced to learn specialized knowledge that is already available. On the contrary, classic RRL methods have great generalization capabilities, they can transfer knowledge across tasks, and they can exploit an existing background knowledge. However, classic RRL methods are not able to cope with the uncertainty of real-world environments. Moreover, they cannot solve complex tasks because they are not able to concisely represent the action-value function. In order to overcome the limits of both DRL and RRL, in recent years neural-symbolic methods for RL has received more and more attention. These methods try to combine modern deep learning techniques with a relational representation of states and actions. The interpretability of the learned solution was not the main focus of RRL, but the interpretability of AI

methods has become a major issue in the last few years. The increasing interest towards this problem has brought attention to *eXplainable AI* (*XAI*) [50]. However, XAI focuses on interpretability and explainability problems, and generalization capability, as well as other interesting properties, are not directly considered. In next chapters, a new neural-symbolic method for RL is proposed. The proposed method is able to learn logic rules from a logical representation of states and actions. Using a logical representation of the policy is important from the perspective of XAI.

Chapter 4

A Survey of Methods for ILP

Inductive Logic Programming (*ILP*) algorithms try to induce logical rules from positive and negative examples, and they represent a good starting point to tackle even *Reinforcement Learning* (*RL*) problems. This chapter presents a comparison between the major neural-symbolic methods for ILP that have been presented in the last 5 years. Neural-symbolic methods for ILP represent a good starting point to solve RL tasks. Therefore, this chapter analyzes the current state-of-the-art to compare the main neural-symbolic approaches to ILP, which can be then used to tackle RL tasks. The chapter is structured as follows. Section 1 introduces the core concepts of first order logic, and it briefly describes ILP. Section 2 presents the most influential neural-symbolic methods for ILP that have been presented in the last 5 years, discussing the main differences and briefly comparing their main characteristics. Section 3 discusses a comparison of the neural-symbolic methods that have been presented in the previous section from the perspective of the interpretability of the induced rules. Section 4 presents a comparison of the neural-symbolic methods that have been presented in Section 2 from the perspective of the reusability of the induced rules. Finally, Section 5 concludes the chapter discussing some open challenges of the field as well as some future research directions.

4.1 Background

Deep Reinforcement Learning (*DRL*) and *Relational Reinforcement Learning* (*RRL*) have complementary strengths and weaknesses. DRL algorithms are very good at treating noisy or erroneous data, and they are normally applied to non-symbolic data, but they do not produce interpretable policies, and they often fail to generalize to unseen situations. On the other hand, RRL algorithms are much more data efficient than DRL methods, and they produce policies that are interpretable, and that allow to generalize to unseen

situations. Unfortunately, RRL methods are often not able to scale to complex tasks, they are typically unsuitable to handle noisy or erroneous data, and they do not normally support the use of non-symbolic data. The characteristics of sub-symbolic methods can prevent the use of these approaches in critical domains, like self-driving vehicles, and they raise several important concerns related to ethical and legal issues. At the same time, the characteristics of symbolic methods can prevent the use of these approaches in complex and real-world applications. Therefore, symbolic and sub-symbolic approaches are normally considered complementary, and the literature is witnessing several attempts at combining them in the so-called *neural-symbolic approaches* (e.g., [51]).

A common approach to solve RL tasks with neural-symbolic algorithms is to adapt neural-symbolic approaches to ILP [9–11]. Inductive logic programming (e.g., [8]) has been studied for more than 30 years with the major goal of delivering effective algorithms to induce logical rules from data. Given sufficient background knowledge expressed as a set of logical rules and a set of initial examples represented as positive and negative facts, the goal of ILP is to derive a logic program that entails all positive examples while rejecting all negative examples.

ILP has several advantages when compared with other machine learning algorithms, as follows:

1. ILP allows using background knowledge, expressed as facts and logical rules, and experts can inject structured knowledge to improve the effectiveness of learning;

2. ILP generates logical rules that—as opposed to, for example, the weights of an artificial neural network—are easily interpretable by humans (e.g., [52, 53]). The generated rules can be used to reason about the learning task and to perform logical deduction;

3. ILP provides algorithms that are normally considered as data efficient because they typically generalize better than other machine learning algorithms (e.g. [15]), which is very important when available data are barely sufficient for a superficial learning; and

4. ILP supports continual and transfer learning because the background knowledge can be continuously extended.

State-of-the-art ILP techniques now provide advanced features, like the induction of recursive rules, that were still considered as huge obstacles a few years ago. Even if traditional ILP gained more and more interest in the last 30 years [53], neural-symbolic approaches to ILP have the potential to represent a turning point for the research on

this subject. Neural-symbolic ILP allows combining the advantages of traditional ILP with recent advancement in deep learning, and it represents a new way to generate reusable and interpretable logic programs from noisy data. This section provides a brief description of ILP to introduce the adopted nomenclature and to fix notation. Interested readers can consult one of the several introductory texts on ILP (e.g. [8]) for more details on the subject.

4.1.1 First-order logic

First-Order Logic (*FOL*) is a formal language in which the major interest is on logical deduction. The syntax of FOL comprises several elements: constant symbols, variable symbols, function symbols, and predicate symbols. Constant symbols, or constants, are normally denoted by words starting with lowercase letters, and they are interpreted as the objects of the domain of discourse. Variable symbols, or variables, are normally denoted by words starting with uppercase letters, and they are interpreted as references to unspecified objects of the domain of discourse. Function symbols, or functions, are normally denoted by words starting with lowercase letters, and they are interpreted as mappings in the domain of discourse from tuples of objects to single objects. Predicate symbols, or predicates, are normally denoted by words starting with lowercase letters, and they are interpreted as relations among objects in the domain of discourse.

In the syntax of FOL, a term can be a constant, a variable, or a function applied to a tuple of terms. For example, if c is a constant then $f(c)$ and $f(f(c))$ are terms, where f is a function symbol. An atom is a tuple $p(t_1, \dots, t_n)$, where p is a predicate symbol applied to a tuple of terms $t_1, \dots, t_n$, while a literal is an atom (positive literal) or a negated atom (negative literal). A (disjunctive) clause is a logical formula expressed as a disjunction of literals. All variables are universally quantified in a clause, often implicitly. A definite clause is nothing but a clause with only one positive literal. In this chapter, rule, clause, and definite clause are used as synonyms, unless explicitly reported. A rule is typically written as:

$$\alpha \Leftarrow \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n, \tag{4.1}$$

where the atoms $\beta_1, \dots, \beta_n$ form the body of the rule, and α is the head of the rule. A definite clause represents a logical consequence: if all the atoms in the body $\beta_1, \dots, \beta_n$ are true, the head atom α is true. A Horn clause is a clause with at most one positive literal. A ground clause (resp. atom) is a clause (resp. atom) that contains no variables. Predicates can be characterized by listing a set of ground atoms, normally called facts, to obtain extensional predicates. Alternatively, predicates can be characterized using

a set of facts and rules to obtain intensional predicates. In order to obtain a ground rule from a definite clause, a variable substitution is needed, which can be defined as $\mu = \{X_1/t_1, \ldots, X_n/t_n\}$. The substitution set μ represents an assignment of terms t_i to variables X_i, a variable substitution is denoted as $c[\mu]$, where c is a generic rule.

There are two main techniques to perform logical deduction: forward chaining and backward chaining. In order to define the two approaches, the concept of entailment is necessary. Given a set of definite clauses C, and a ground atom γ, C *entails* γ, or $C \models \gamma$, if γ is a logical consequence of C. In order to perform logical deduction, forward chaining requires to deduce the logical consequences of the rules C for a specified number of steps T. Then, it verifies if the atom γ is included in the set of ground atoms that are produced. Given a set of clauses C, and a ground atom γ, the set of logical immediate consequences of C applied to a set of ground atoms X can be defined as:

$$icn_C(X) = X \cup \{\gamma \mid \gamma \Leftarrow \gamma_1, \ldots, \gamma_m \in ground(C), \bigwedge_{i=1}^{m} \gamma_i \in X\}, \tag{4.2}$$

where $ground(C)$ is the set of ground rules of C. Using the previous definition, it is possible to define the logical consequences after T steps as:

$$scn(C) = \bigcup_{i \geq 0}^{T} S_{C,i}, \tag{4.3}$$

where $S_{C,i}$ are the immediate consequences derived at step i:

$$S_{C,0} = \{\} \qquad S_{C,i+1} = icn_C(S_{C,i}) \tag{4.4}$$

Now, if $\gamma \in scn(C)$, then $C \models \gamma$. Backward chaining does the opposite, and it tries to find a rule $\alpha \Leftarrow \beta_1, \cdots, \beta_n \in C$, and a variable substitution μ, such that $\alpha[\mu] = \gamma$. Each atom in the body of the rule becomes a new sub-goal, and the algorithm tries to recursively prove each sub-goal performing the same operation until all the atoms in the bodies of the selected rules are proved.

4.1.2 ILP

This chapter takes into consideration a specific form of ILP, which is defined as *learning from entailment*. The literature presents other definitions of ILP that can be useful in other learning settings [53]. An ILP task can be defined as a tuple $\langle B, \mathcal{N}, \mathcal{P} \rangle$, where B is a set of ground atoms and background clauses, $\mathcal{N}$ and $\mathcal{P}$ are sets of negative and positive ground atoms, respectively. The goal of an ILP task is to induce a set of

clauses that entails all elements of $\mathcal{P}$ and rejects all elements of $\mathcal{N}$. More formally, a solution to an ILP task is a set of definite clauses C where:

$$\begin{aligned} B, C &\models \gamma \quad \forall \gamma \in \mathcal{P} \\ B, C &\not\models \gamma \quad \forall \gamma \in \mathcal{N} \end{aligned} \tag{4.5}$$

The solution to an ILP task normally requires searching in the space of clauses, which grows exponentially with the number of available constants, functions, and predicates. Therefore, ILP methods normally restrict the space of FOL clauses to definite clauses to make ILP tasks feasible.

For example, the following ILP task adapted from [15] can be used to induce the predicate $even(X)$, which is expected to hold for every even natural number X. If the set of constants is $C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the considered task can be defined as:

$$\begin{aligned} B &= \{zero(0), succ(0, 1), succ(1, 2), succ(2, 3), succ(3, 4), succ(4, 5)\} \\ \mathcal{P} &= \{even(0), even(2), even(4)\} \\ \mathcal{N} &= \{even(1), even(3), even(5)\} \end{aligned} \tag{4.6}$$

where B is the background knowledge that characterizes zero and the successor predicate *succ*. Using one of the available ILP methods, the following set of definite clauses can be induced:

$$\begin{aligned} even(X) &\Leftarrow zero(X) \\ even(X) &\Leftarrow succ2(Y, X) \wedge even(Y) \\ succ2(X, Y) &\Leftarrow succ(X, Z) \wedge succ(Z, Y) \end{aligned} \tag{4.7}$$

where $succ2(X, Y)$ is an invented predicate that is interpreted as $X = Y + 2$. The obtained predicate clearly entails all the elements of $\mathcal{P}$, rejects all the elements of $\mathcal{N}$, and it is able to generalize to unforeseen constants like 6 and 8.

4.2 Neural-symbolic ILP

This section presents some of the most relevant neural-symbolic ILP methods that have been proposed in the last five years. The discussed methods were selected starting from recent surveys on ILP and neural-symbolic methods [51–55], which were complemented with the inclusion of papers that the surveys cite, possibly indirectly. Interested readers can consult the mentioned surveys to gain a broader understanding of the subject. In particular, [54] performs a comparison between neural-symbolic and statistical-relational approaches. [52] discusses the recent literature from an *eXplainable Artificial Intelligence* (*XAI*) perspective, and it classifies notable works from the perspective of

the integration between symbolic and sub-symbolic approaches. [51] compares recent literature on the basis of the representations of data, and it also classifies discussed papers according to which part, either the symbolic part or the sub-symbolic part, is the most relevant. [53] focuses on traditional ILP and briefly discusses some of the main neural-symbolic approaches. Finally, [55] compares relevant papers from major conferences that target neural-symbolic algorithms according to several interesting features.

Most neural-symbolic methods for ILP use some form of program templates, which can be generally defined as a set of hyper-parameters that guide the rule generation. In fact, as discussed previously, rule generation requires a large search space, and in the last few years, many solution have been proposed. Program templates are not an ideal solution to this problem but they are interesting because they require the user to specify only a few parameters, as opposed to other techniques that require the user to manually specify the structure of the generated rules, e.g., meta-rules [56].

Neural-symbolic methods for ILP are not always focused on a specific logic language. In this chapter, different approaches induce programs written in different subset of FOL. An interesting logic language is Datalog, which imposes some restrictions on the form of the programs. Many Datalog dialects are discussed in the literature. In order to be general, this chapter defines Datalog programs as a lists of definite clauses and facts in which function symbols are not allowed. In order to provide an introductory view on this subject, 4.4 presents a comparison between the neural-logic methods for ILP that induce programs written in Datalog dialects. Datalog dialects are interesting because they are sufficiently expressive to solve complex problems and, at the same time, these languages are decidable, and therefore computations always terminate. Therefore, only Datalog dialects have been considered because they have good expressive power and, at the same time, they impose a considerable limit on the space of induced programs.

It is worth noting that some interesting approaches do not produce logical rules that are sufficiently general, and therefore they are not discussed in this work. For example, NLIL [57, 58] can be used to induce logical rules that follow a rigid chained structure, which is less expressive than Datalog. Finally, DeepProbLog [59] extends ProbLog [60] to allow defining neural predicates in which deep neural networks are used to implement predicates capable to process noisy or erroneous data.

4.2.1 δILP

In δILP [15], rules are generated from a program template and then tested using the training data. In order to cope with noisy or erroneous data, δILP uses a continuous relaxation of the truth value of each rule, which is then associated to a weight that

represents the probability of the rule to be part of the induced program. Each element in $\mathcal{P}$ and $\mathcal{N}$ is associated with a value in $[0, 1]$ that represents the truth value of the element. δILP reinterprets the ILP task as a binary classification task. In particular, in order to choose the correct value for the weights of rules, δILP introduces a differentiable implementation of deduction to predict, for each rule, the truth value of the atoms that are randomly selected from $\mathcal{P}$ and $\mathcal{N}$. Training the model of δILP requires computing the expected truth value of positive and negative examples, which is obtained performing a prefixed amount of forward chaining steps and then applying the generated rules to the facts in the background knowledge.

As usual for ILP tasks, the induction of rules from data requires searching in a search space whose size grows exponentially with the number of constants and predicates in B. In order to reduce the size of the search space, δILP generates rules from a program template, which is defined as follows:

$$\Pi = \langle \text{Pred}_a, \text{arity}_a, (\tau_p^1, \tau_p^2), T \rangle, \tag{4.8}$$

where Pred_a and arity_a identify a set of auxiliary intensional predicates, which are invented and used to define the target predicate, together with their arities. $T \in \mathbb{N}$ is the number of forward chaining steps to perform, and τ_p^1 and τ_p^2 are two rule templates that define how each rule should be generated for each intensional predicate $p \in \text{Pred}_i$, where Pred_i is the set of intensional predicates. In order to further constraint the generation of rules, each rule template is defined as a tuple:

$$\tau = \langle v, int \rangle, \tag{4.9}$$

where $v \in \mathbb{N}$ is the number of free variables in the rule, and $int \in \{0, 1\}$ specifies whether intensional predicates are allowed or not. Since there are only two rule templates, each generated intensional predicate, either auxiliary or target, is defined by exactly two rules.

For each rule template, δILP generates a set of rules $cl(\tau)$ using the following further restrictions:

1. Constants are not allowed in generated rules;

2. Predicates of arity higher than three are not allowed in generated rules;

3. Each generated rule must contain exactly two atoms in its body;

4. All variables that appear in the head of a generated rule must appear also in its body;

5. Two rules that differ only in the order of the atoms in their bodies are not both allowed in the set of generated rules; and

6. An atom must not be used at the same time in the head and in the body of a generated rule.

For each generated predicate p, δILP defines a matrix $W_p \in \mathbb{R}^{|cl(\tau_p^1)|\times|cl(\tau_p^2)|}$ that contains the weights associated with the rules that define the predicate. Each matrix W_p has $|cl(\tau_p^1)|$ rows and $|cl(\tau_p^2)|$ columns, where $|cl(\tau_p^i)|$, with $i \in \{1,2\}$ is the number of rules generated from the corresponding rule template τ_p^i. In order to transform W_p into a probability distribution, a softmax function is applied, obtaining $W_p^* \in [0,1]^{|cl(\tau_p^1)|\times|cl(\tau_p^2)|}$ defined as:

$$W_p^*[j,k] = \frac{e^{W_p[j,k]}}{\sum_{j',k'} e^{W_p[j',k']}} \tag{4.10}$$

Each element $W_p^*[j,k]$ represents the probability that rules j and k are the correct rules to define predicate p. The set of W_p^* matrices is defined as W.

δILP uses a continuous relaxation of the truth value of ground atoms. In particular, given a set of n ground atoms G, and a ground atom $\gamma \in G$, a valuation is a vector in $[0,1]^n$ which performs a mapping between each element $\gamma_i \in G$ to $\mathbb{R}$. In order to train the weight matrices W_p, δILP takes, for each pair of rules, T forward chaining steps, and it tries to predict the correct truth value for each element of the training set. In particular, given the example (γ, λ), where γ is an atom and λ is the associated label, the entire training set is defined as:

$$\Lambda = \{(\gamma, 1) \mid \gamma \in P\} \cup \{(\gamma, 0) \mid \gamma \in N\}. \tag{4.11}$$

In summary, δILP samples a training example (γ, λ) from Λ and, using the background knowledge B and the language L, which consists of the constants and the extensional predicates together with the target predicate, it tries to predict the following probability:

$$P(\lambda \mid \gamma, W, \Pi, L, B) = f_{\text{extract}}(f_{\text{infer}}(f_{\text{convert}}(B), f_{\text{generate}}(\Pi, L), W, T), \gamma), \tag{4.12}$$

where $f_{\text{extract}}, f_{\text{infer}}, f_{\text{convert}}, f_{\text{generate}}$ are auxiliary functions.

The computation of $P(\lambda \mid \gamma, W, \Pi, L, B)$ can be summarized in the following four steps:

1. The function $f_{\text{convert}} : 2^G \rightarrow [0,1]$ is used to obtain the expected truth value of each ground atom:

$$f_{\text{convert}}(B) = y \quad \text{where} \quad y[i] = \begin{cases} 1 & \text{if } \gamma_i \in \text{B} \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

where γ_i is the i-th element of G for $i = 1, \ldots, n$.

2. Then, a set of rules is generated by applying f_{generate} to the language L and program template Π:
$$f_{\text{generate}}(\Pi, L) = \{cl(\tau_p^i) \mid p \in \text{Pred}_i, i \in 1, 2\}. \tag{4.14}$$

3. Then, T forward chaining steps are performed to compute the predicted truth value for each example in Λ using $f_{\text{infer}} : [0,1]^n \times C \times W \times \mathbb{N} \to [0,1]^n$, which requires the generated rules, the initial truth values, and the current rule weights.

4. Finally, the function $f_{\text{extract}} : [0,1]^n \times G \to [0,1]$ is used to extract the predicted probability of λ given a specific atom γ:
$$f_{\text{extract}}(x, \gamma) = x[index[\gamma]], \tag{4.15}$$
where $index : G \to \mathbb{N}$ is a function that retrieves the unique index of each ground atom.

δILP makes use of the computed probability to compute a loss as the cross-entropy of the correct label with regard to the predicted label:

$$loss = -\mathbb{E}_{(\alpha,\lambda)\sim\Lambda}[\lambda \cdot \log p(\lambda|\gamma, W, \Pi, L, B) + (1-\lambda) \cdot \log(1 - p(\lambda|\gamma, W, \Pi, L, B))], \tag{4.16}$$

and then it adjusts the corresponding rule weights accordingly by using the stochastic gradient descent algorithm. At the end of the training, the weights of rules can be used to extract the most appropriate pair of rules to define the target and the auxiliary predicates that solve the ILP problem.

The logical deduction of δILP is based on a set of functions $F_c : [0,1]^n \to [0,1]^n$, defined for each rule c. Each function F_c takes as input a valuation that represents the initial truth value of the ground atoms, and it computes a valuation that represents the new truth value obtained from the application of the rule c. Assuming the existence of a mapping between each ground atom and a unique index, for each F_c a three-dimensional vector $X_c \in \mathbb{N}^{n \times w \times 2}$ is computed before the training phase, where n is the number of ground atoms. The shape of X_c is also determined by w, which represents the maximum number of pairs of ground atoms that logically entail each ground atom. Formally, X_c can be defined using x_k as the set of index pairs that justify the k-th ground atom:

$$\begin{aligned} x_k &= \{(a, b) \mid \text{satisfies}_c(\gamma_a, \gamma_b) \wedge head_c(\gamma_a, \gamma_b) = \gamma_k\} \\ X_c[k, m] &= \begin{cases} x_k[m] & \text{if } m < |x_k| \\ (0, 0) & \text{otherwise} \end{cases} \end{aligned} \tag{4.17}$$

Intuitively, X_c allows to represent, for each ground atom γ_k, the set of ground-atom pairs whose logical consequence is the considered atom. It is worth noting that X_c contains a portion of unused space because the number of pairs that justify each ground atom is variable, and the maximum number of pairs is used for the vector construction. In order to solve this problem, the falsum atom $\perp$ is included in G. In case of unused space, δILP inserts the pair $(0, 0)$, which is mapped to the atom pair $(\perp, \perp)$.

The structure X_c is built to define one step of forward chaining deduction for rule c. The actual deduction is made during the training phase as followings. δILP takes $X_1, X_2 \in \mathbb{N}^{n\times w}$, two slices of X_c that represent the first and second elements in each pair, respectively:

$$X_1 = X_c[_, _, 0] \qquad X_2 = X_c[_, _, 1] \tag{4.18}$$

Then, δILP defines two matrices $Y_1, Y_2 \in [0, 1]^{n\times w}$ that represent the vectors containing the actual truth values of a valuation a referenced by X_1, X_2. The computation of Y_1, Y_2 is made by translating each index to the corresponding truth value using the function $gather_2 : \mathbb{R}^a \times \mathbb{N}^{b\times c} \rightarrow \mathbb{R}^{b\times c}$:

$$gather_2(x, y)[i, j] = x[y[i, j]] \tag{4.19}$$

$$Y_1 = gather_2(a, X_1) \qquad Y_2 = gather_2(a, X_2) \tag{4.20}$$

Then, δILP defines a vector $Z_c \in [0, 1]^{n\times w}$ that is computed by performing the element-wise multiplication of Y_1 and Y_2:

$$Z_c = Y_1 \odot Y_2 \tag{4.21}$$

Finally, δILP defines F_c as:

$$F_c(a) = a' \quad \text{where} \quad a'[k] = max(Z[k, _]) \tag{4.22}$$

δILP performs the computation of F_c taking the maximum value of $Z[k, _]$ for the atom γ_k because $Z[k, _]$ represents the vector of fuzzy conjunctions of all the pairs of ground atoms that contribute to the truth value of γ_k. Therefore, taking the maximum truth value from all the pairs of atoms implements a fuzzy disjunction.

Once the forward chaining step for a single clause c is defined, δILP defines a step for a pair of clauses of predicate p as:

$$C_p^{j,k}(a) = x \quad \text{where} \quad x[i] = max(F_p^{1,j}[i], F_p^{2,k}[i]) \tag{4.23}$$

where $F_p^{i,j}$ performs a forward chaining step for the j-th clause of the i-th rule template. In the same way of F_c, $C_p^{j,k}$ computes the resulting truth value by taking the maximum value of the two clauses of each rule template for each ground atom.

Using the functions $C_p^{j,k}$, it is possible to define a sequence of T forward chaining steps. The initial valuation is defined by f_{convert}:

$$a_0[x] = \begin{cases} 1 & \text{if } \gamma_x \in \mathrm{B} \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

In order to define the step $t+1$, it is useful to define the result valuation b_t as:

$$b_t = \sum_{p \in \mathrm{Pred}_i} \sum_{j,k} C_p^{j,k} \cdot W_p^*[j,k] \tag{4.25}$$

b_t is defined by applying all the possible pair of rules that jointly defines a predicate p, weighting the results by W_p^*. The resulting valuations are then summed for each predicate p because they are disjointed. The result of the $t+1$ is then computed as the probabilist sum between the valuation at step t and the resulting valuation b_t:

$$a_{t+1} = a_t + b_t - a_t \cdot b_t \tag{4.26}$$

The architecture of δILP allows implementing logical deduction that handles uncertainty because the resulting truth value for each ground atom is a real value in $[0,1]$. The role of each rule in the deduction process is adjusted with the corresponding weight, which is continuously updated during training. Therefore, the algorithm increases the weights corresponding to the clauses that entail the positive examples, and it does the opposite with the clauses that entails the negative examples. As discussed previously, the architecture can be described using four functions. The computation of X_c is non-differentiable, and it is therefore performed before the training phase. f_{extract} and f_{infer} are instead differentiable operations. In particular, the function $gather_2$ allows f_{infer} to be differentiable with respect to the weights, making the training phase feasible.

Despite the interesting results of δILP with respect to traditional ILP methods, δILP suffers from two major problems:

1. It stores the weights for every generated pair of rules, which makes the method unusable in difficult tasks, with many background and auxiliary predicates, because the number of generated rules may be very large.

2. The imposed restrictions on the generation of rules makes δILP not suitable in complex tasks because more flexible methods can produce more compact and effective rules.

4.2.2 NTPs

Neural Theorem Provers (*NTPs*) [16–18] are a class of differentiable end-to-end provers for theorems formulated as queries to a knowledge base. They implement a Prolog-style backward chaining algorithm that makes use of differentiable unification among sub-symbolic representations of logical entities. In particular, given a query, NTPs use three operators: AND, OR, and UNIFY. Here, the first paper that introduces this method [16] is used for the presentation of the method.

The UNIFY operator is similar to the one used by Prolog to verify the exact structural match among predicates and among terms. In NTPs, the UNIFY operator verifies the similarity among vector representations of symbols. After an application of the UNIFY operator, the method updates a substitution set defined as $\mu = \{X_1/t_1, \ldots, X_n/t_n\}$. Starting from a target query, μ allows applying the rules in the knowledge base to the constants in the query. The Prolog-style backward chaining algorithm applies the OR operator on each rule trying to unify the query with the head of a definite clause. If unification succeeds, the method uses the AND operator to jointly prove all the atoms that are part of the body of the selected rule. Finally, the AND operator performs a substitution of the variables of the selected atoms, and it tries to compute their truth values applying, one more time, the OR operator.

Each operator receives atoms, rules, and a proof state, and it returns a proof state list. A proof state is a tuple $\langle \delta, \rho \rangle$, where δ is the current substitution set, and ρ is an artificial neural network that computes a real value that represents a partial proof score. ρ is built during training, but it can be used in both training and testing phases to prove different target queries. When the proof algorithm ends, the method selects the proof state that maximizes the proof score among the ones built using the OR operator applied on each rule of the knowledge base.

While the OR operator and the AND operator are defined ordinarily, UNIFY is the most characterizing operator of this method. Instead of checking the equality between two non-variable symbols, the UNIFY operator compares their vector representations using a *Radial Basis Function* (*RBF*) kernel [61]. The use of a RBF kernel allows the application of the rules of the knowledge base to symbols that are apparently different, but that share a common meaning. An example of the benefits of using a RBF kernel is to match apparently different predicates like *grandPa* and *grandFather*.

In order to train the model, NTPs compute a loss function. In particular, they obtain invalid ground atoms $[s, \hat{i}, j]$, $[s, i, \hat{j}]$, $[s, \tilde{i}, \tilde{j}]$ from atoms $[s, i, j]$ that are present in the knowledge base $\mathcal{K}$. These corrupted atoms must not be present in $\mathcal{K}$, and $\hat{i}$, $\hat{j}$, $\tilde{i}$ and $\tilde{j}$ are

constants. The corrupted atoms represent the negative examples in $\mathcal{N}$, while the known atoms represents the positive examples in $\mathcal{P}$. NTPs compute the loss as the negative log-likelihood of the proof success score, as follows:

$$loss = \sum_{([s,i,j],y) \in \mathcal{T}} -y \log(ntp_\theta^{\mathcal{K}}([s,i,j],d)_\rho) - (1-y)\log(1 - ntp_\theta^{\mathcal{K}}([s,i,j],d)_\rho), \quad (4.27)$$

where y is the target proof score (either 0 or 1), and $\mathcal{T}$ is the set of known and invalid atoms. The function $ntp_\theta^{\mathcal{K}}(X,d)_\rho$ computes the overall success score of proving a goal X using a maximum proof depth d, and it is parameterized with respect to θ, which is a matrix that encodes the approximate representations of non-variable symbols. The proof success score ρ is explicitly written in the loss function because, in the training phase, when proving a known atom, the substitution set δ is not considered for optimization purposes. The original paper [16] presents also a variant of NTP called NTPλ, which employs a modified loss and performs better than NTP in many proposed tasks.

NTPs are not designed as ILP methods, but they allow performing logical deductions to extend existing knowledge bases. In order to perform rule induction, NTPs use a form of *meta-rules*, which are rules where non-variable symbols are not specified. The goal of the method is to learn their representations from data. For example, the transitivity rule, $r(X,Y) \Leftarrow f(X,Z) \wedge g(Z,Y)$, where r, f, g are predicates that are not part of the knowledge base, is a meta-rule. Meta-rules are a form of templates that allow the user to describe the structure of induced rules using a specific syntax, like, for example [16]:

$$\begin{aligned} &n_1 \ \#m_1(X,Y) \text{ :- } \#m_2(Y,X) \\ &n_2 \ \#m_3(X,Y) \text{ :- } \#m_4(X,Z),\ \#m_5(Z,Y) \end{aligned} \quad (4.28)$$

where n_i represents how many times meta-rule i is instantiated, while each m_i is a placeholder for a predicate name, and each placeholder is meant to be substituted with a different predicate name. Note that, in general, NTPs seem to be more scalable than δILP, but they offer less flexibility because they require to explicitly specify the structure of the generated rules.

4.2.3 ILP$_{\text{Camp}}$

Campero et al. propose in [21] a neural-symbolic ILP method, which is informally called ILP$_{\text{Camp}}$ here for the sake of clarity. ILP$_{\text{Camp}}$ follows an approach similar to NTPs because predicates are represented as vectors of real numbers but, in this case, forward chaining is used to obtain new facts. In the original paper [21], Campero et al. describe the algorithm for binary predicates only. The following is a generalization

of the implementation discussed in [21] for predicates of higher arity. ILP_{Camp} stores a list of facts, each of which is represented as a tuple $(\theta_p, c_1, \ldots, c_n, \upsilon)$, where θ_p is a vector representing predicate p, which is shared among all of its ground atoms, c_i are the constants associated with the fact, and $\upsilon \in [0, 1]$ represents the fuzzy truth value of the fact. ILP_{Camp} assigns a truth value of 1 to ground atoms that are part of the background knowledge, and it stores a set of vectors of real numbers Pred, which are randomly initialized. Pred includes predicates that are defined as part of the task together with the auxiliary predicates that the user expects to be necessary. As NTPs, ILP_{Camp} generates rules using meta-rules, each of which is expressed as, for example:

$$F(X, Y) \Leftarrow F(X, Z), F(Z, Y), \tag{4.29}$$

where F is a symbol that stands for a generic predicate, and its underlying representation is learned during the training phase. Internally, each meta-rule can be represented by:

$$((\theta_h, v_{h_1}, \ldots, v_{h_n}), (\theta_{b1}, v_{b1_1}, \ldots, v_{b1_n}), \ldots, (\theta_{bn}, v_{bn_1}, \ldots, v_{bn_n})), \tag{4.30}$$

where $\theta_h, \theta_{b1,} \ldots, \theta_{bn}$ are vectors of real numbers representing the predicates in the head and the body of the meta-rule, while each v_i is the name of an used variable. In order to perform a single forward chaining step, the method generates an admissible grounding set that is chosen by verifying the structure of the meta-rule and by replacing variables if the structure of the meta-rule is respected. For example, using the rule in 4.29 and supposing a set of constants $\{a, b, c\}$, the rule can be applied to ground atoms $\{F(a, b), F(b, c)\}$ because they respect the structure of the meta-rule. On the contrary, other ground atoms, such as $\{F(a, b), F(a, c)\}$, do not respect the structure of the meta-rule, and therefore they cannot be used.

Using each meta-rule, ILP_{Camp} deduces new facts starting from each pair of ground atoms that are admissible. For each meta-rule and for each pair of atoms, the algorithm generates the atoms $f_1, \ldots, f_n$, one for each predicate in Pred. The internal structure of the new ground atoms is $(\theta_o, c_{o_1}, \ldots, c_{o_n}, \upsilon_o)$. Constants c_{o_i} are obtained from the pairs of predicates used in the substitution phase. The value of υ_o for each new fact is computed by multiplying the values of $f_1, \ldots, f_n$ with the cosine similarity of the vectors associated with the predicates in the meta-rule and in the existing facts, as follows:

$$\upsilon_o = \cos(\theta_h, \theta_o) \cdot \cos(\theta_{b1}, \theta_{f_1}) \cdot \cdots \cdot \cos(\theta_{bn}, \theta_{f_n}) \cdot \upsilon_{f_1} \cdot \ldots \cdot \upsilon_{f_n}. \tag{4.31}$$

When ILP_{Camp} finds a new ground atom, it stores the associated representation in the list of facts, if it is not already in the list. Otherwise, ILP_{Camp} simply updates the representation of the new fact in the list computing the maximum between the new truth

value v_o and the previous one. After several deduction steps, ILP_{Camp} generates new facts from existing facts. The trainable parameters are those associated with defined predicates and those associated with the meta-rules θ_h, θ_{bi}. Using positive and negative examples in $\mathcal{P}$ and $\mathcal{N}$, respectively, ILP_{Camp}can train θ_h and θ_{bi}, computing binary cross-entropy to predict the truth value of the examples and to generalize to unforeseen constants. Both the number of forward chaining steps and the number of auxiliary predicates are defined as hyper-parameters of the method.

ILP_{Camp} reports interesting results when compared to δILP and NTPs. ILP_{Camp} reports slightly better performance than δILP, and it exhibits performance that are comparable to NTPs, except for one specific task where NTPs performance is much worse, for the three problems taken into consideration. However, ILP_{Camp} has some drawbacks. The use of forward chaining implies that the generation of rules requires an increasing number of facts as the training progresses. Therefore, ILP_{Camp} cannot scale well as the size of the background knowledge and the number of meta-rules increases. Conversely, NTPs start from the target query and they try to prove it using backward chaining, which does not require to generate too many atoms at each time step. Compared to δILP, meta-rules represent a less flexible solution than program templates because meta-rules require a human expert to manually specify the structure of the generated rules.

4.2.4 dNL-ILP

dNL-ILP [19, 20] associates a trainable weight with each atom that can be part of the generated rules. Note that this approach is different from δILP because δILP associates weights with rules while dNL-ILP associates weights with atoms. dNL-ILP defines a particular neural network to model the structure of a *Conjunctive Normal Form* (*CNF*) or of a *Disjunctive Normal Form* (*DNF*) formula, and the nodes of the neural network implement fuzzy operators defined as follows:

$$\bar{x} = 1 - x \qquad x \wedge y = xy \qquad x \vee y = 1 - (1 - x)(1 - y). \tag{4.32}$$

dNL-ILP associates a truth value in $[0, 1]$ to each atom, just like δILP, and each node of the neural network receives an input vector $x \in [0, 1]^n$. This algorithm defines a membership weight $m_i \in [0, 1]$ for each input value x_i, which is used to include or exclude the input atom from the generated rule. The neural network is composed of tree types of nodes: AND, OR, and NOT. The first two types of nodes implement functions f_{AND} and f_{OR} defined as:

$$f_{\text{AND}}(x) = \prod_{i=1}^{n} (1 - m_i(1 - x_i)) \tag{4.33}$$

$$f_{\text{OR}}(x) = 1 - \prod_{i=1}^{n}(1 - m_i x_i) \tag{4.34}$$

The neural network is built alternating AND and OR layers, and the order of layers defines the form of the generated rules. Using a first layer of AND nodes followed by a second layer of OR nodes, the algorithm produces DNF formulae. On the contrary, CNF formulae are produced if a first layer of OR nodes is followed by a second layer of AND nodes.

dNL-ILP requires the user to specify a program template, which is defined as:

$$\Pi = \langle \text{Pred}, \text{arity}_{\text{Pred}}, Nr_{\text{Pred}}, Nv_{Nr_{\text{Pred}}} \rangle, \tag{4.35}$$

where Pred defines the target and the auxiliary predicates, $\text{arity}_{\text{Pred}}$ their respective arities, Nr_{Pred} is the number of rules that define each predicate, and $Nv_{Nr_{\text{Pred}}}$ is the number of variables in the body of each generated rule. For each predicate $p \in \text{Pred}$ and each forward chaining step t, the algorithm defines a value vector $X_p^t \in [0,1]^n$ that contains the fuzzy truth values of every atom resulting from the application of p. If G_p is the set of ground atoms associated with a predicate p, and $\gamma \in G_p$, the initial fuzzy vectors X_p^0 are defined as follows:

$$X_p^0[\gamma] = \begin{cases} 1 & \text{if } \gamma \in B \\ 0 & \text{otherwise} \end{cases} \tag{4.36}$$

dNL-ILP starts with fuzzy values $X_p^0[\gamma]$, and it performs T forward chaining steps, obtaining the values $X_p^T[\gamma]$ that are interpreted as the conditional probability of each truth value with respect to the model parameters. During the training, dNL-ILP computes a cross-entropy loss between $X_p^T[\gamma]$ and the truth values associated with training examples.

For the sake of simplicity, only generated formulae in DNF are considered now. Let F_p^i be the conjunctive function represented by the neural network described previously for predicate p and rule i, and let $\Theta_p^i(\gamma)$ be the set of substitutions among constants and variables that result in the atom γ. Let $\mathbb{I}_p^i$ be the set of possible atoms for the predicate p and the rule i, which is defined as:

$$\mathbb{I}_p^i = \bigcup_{p' \in \text{Pred}} \mathbb{T}(p', V_p^i) \quad \text{where} \quad \mathbb{T}(p, V) = \{p(arg) \mid arg \in Perm(V, arity(p))\}, \tag{4.37}$$

where V_p^i is the set of all variables of predicate p and rule i, $Perm(s, n)$ generates all the permutations of length n from the set s. Finally, $arity(p)$ returns the arity of predicate p. A single forward chaining step is defined as:

$$X_p^{t+1}[\gamma] = F_{am}(X_p^t[\gamma], K(\gamma)) \quad \text{where} \quad K(\gamma) = \bigvee_i \bigvee_{\theta \in \Theta_p^i(\gamma)} F_p^i(\mathbb{I}^i|_\theta) \tag{4.38}$$

where F_{am} is the fuzzy disjunction function introduced in 4.32.

In summary, for each predicate and rule, dNL-ILP generates a list of every possible atom that can be part of the generated formula, i.e., $\mathbb{I}_p^i$. The algorithm then perform T forward chaining steps to continuously update the values $X_p^t[\gamma]$, which are computed using the disjunction function of the expected truth value of all the possible substitutions of constants that would result in the atom γ. At each forward chaining step, dNL-ILP updates the membership weights m_i, and, at the end of the training phase, dNL-ILP uses the obtained membership weights to extract the logical rules from the network.

It is worth noting that dNL-ILP was subsequently extended to solve RL tasks [11]. Unfortunately, in the original papers [19, 20], only a comparison with respect to δILP is performed. Another performance comparison is presented in [10], which is summarized here. The comparison presented in [10] confronts the performance of DLM, δILP, and dNL-ILP. The two papers [10, 19] report different results: in [19], dNL-ILP performs better than δILP in all the tasks taken into consideration. On the other hand, the results reported by [10] show that δILP performs better than dNL-ILP in almost all the task, which include some of the tasks discussed in [19]. Therefore, these results are contradictory, and further tests are needed to understand the real performance of these methods. In principle, dNL-ILP should be more memory efficient than δILP because it associates weights directly to atoms instead of to entire rules. Interestingly, [10] reports that both δILP and dNL-ILP have memory consumption problems, and both methods fail to complete one task (which is different for each method). In conclusion, dNL-ILP is interesting because it does not require the user to specify the structure of the final solution with meta-rules. As δILP, dNL-ILP makes use of program templates to guide the search for a solution, and it is, in principle, more memory efficient, but more tests are required to understand the real performance of this method. It is worth noting that dNL-ILP requires the user to define the structure of the network, which is not required by the other previously discussed methods.

4.2.5 DLM

Differentiable Logic Machines (*DLM*) [10] is based on another neural-symbolic method called *Neural Logic Machines* (*NLM*) [62]. Both methods are not explicitly designed to solve ILP problems, but they have proved to perform well also for ILP tasks. NLM uses deep neural networks designed to simulate forward chaining, inducing new grounded predicates from a vector representing the truth value of the initial ground predicates. Despite performing relatively well, NLM does not produce interpretable solutions for ILP tasks. DLM tries to solve this problem introducing fuzzy operators, which allows to

produce interpretable and reusable rules.

As NLM, DLM defines a neural architecture that is organized into a grid of logical modules structured in L column and B rows. The first column of the network takes vectors representing the input predicates. Each predicate p of arity b is represented by a vector P of shape (b, m), where m is the number of constants. Each value $P[i_1, \ldots, i_b]$ represents the probability of the truth value of the corresponding ground atom.

The row of a module defines the arity of its input predicates. For example, a module at $[2, i]$ takes as input predicates of arity 2, for $i \in \{1, \ldots, L\}$. Let P_b^l and Q_b^l be the output and input predicates of a module of arity b at column l, respectively. A predicate representing the negation of an existing predicate is obtained applying the function $f(x) = 1 - x$ to each component of the corresponding vector representation. The set of negated predicates is obtained from Q_b^l, and it is denoted as R_b^l. In order to add flexibility to the architecture, DLM allows preserving a predicate from a column to the next, extending Q_b^l and R_b^l with $True$ and $False$ symbols.

Each module defines new predicates P_b^l from Q_b^l using fuzzy conjunction and fuzzy disjunction. The user can specify the number of input predicates to be used to define the output predicates, which is denoted as n_A. Formally, the fuzzy conjunction is defined as:

$$P_b^l = \left(\sum_{P_1 \in Q_b^l} w_{P_1} P_1 \right) \odot \cdots \odot \left(\sum_{P_{n_A} \in Q_b^l} w_{P_{n_A}} P_{n_A} \right), \tag{4.39}$$

where weights $w_{P_1}, \ldots, w_{P_{n_A}} \in [0, 1]$ are the training variables, and $\odot$ is the element-wise product. The fuzzy disjunction is defined as:

$$P_b^l = Q_1 + Q_2 - Q_1 \odot Q_2 \quad \text{where} \quad Q_i = \sum_{P_i \in Q_b^l} w_{P_i} P_i. \tag{4.40}$$

The weights w_{P_i} represents the probability that predicate P_i is the correct one to define the output predicate. Therefore, DLM constraints the value of the weights such that $\sum_{P_i \in Q_b^l} w_{P_i} = 1$. DLM enforces this constraint using a Gumbel-sofmax function [63], which is also employed to avoid early convergence to local optima during training. The Gumbel-softmax function is defined as:

$$w_{P_i} = \frac{e^{\frac{G_P + \theta_P}{\tau}}}{\sum_{P' \in Q_b^l} e^{\frac{G_{P'} + \theta_{P'}}{\tau}}}, \tag{4.41}$$

where G_P are samples from a Gumbel distribution, θ_P are the underlying trainable weights, and τ is the temperature hyper-parameter, which is decreased during training to obtain more interpretable solutions. In order to compute the output predicates, each

module defines $n_O/2$ predicates with fuzzy conjunction and $n_O/2$ predicates with fuzzy disjunction. In particular, both operations are defined on n_A ground atoms that are equally taken by $\{True\} \cup Q_b^l$, $\{False\} \cup Q_b^l$, $\{True\} \cup R_b^l$, and $\{False\} \cup R_b^l$.

Each module is connected to the modules on the previous column of the network. Input predicates Q_b^l are obtained transforming the output predicates resulting from the previous column using three operations:

- *Expansion*: a predicate with arity $b+1$ is obtained from another predicate with arity b:
$$\hat{P}(X_1, \cdots, X_{b+1}) := P(X_1, \cdots, X_b). \tag{4.42}$$
- *Reduction*: a predicate with arity b is obtained from another predicate with arity $b+1$ using either existential and universal quantifiers:
$$\check{P}(X_1, \cdots, X_b) := \exists X_{b+1} P(X_1, \cdots, X_{b+1}), \tag{4.43}$$
$$\check{P}(X_1, \cdots, X_b) := \forall X_{b+1} P(X_1, \cdots, X_{b+1}). \tag{4.44}$$
This operation is implemented using maximum and minimum operators for existential and universal quantifiers, respectively.
- *Permutation*: a predicate with arity b is obtained permuting the arguments of another predicate with the same arity:
$$\forall \sigma \in S_b,\ P_\sigma(X_1, \cdots, X_b) := P(X_{\sigma(1)}, \cdots, X_{\sigma(b)}), \tag{4.45}$$
where S_b is the set of all permutations of $\{1, \ldots, b\}$.

Now, W_b^l can be formally defined as:

$$Q_b^l = \{P_\sigma \mid P \in P_b^{l-1} \cup \hat{P}_{b-1}^{l-1} \cup \check{P}_{b+1}^{l-1}, \sigma \in S_b\}, \quad \hat{P}_0^l = \varnothing, \quad \check{P}_{B+1}^l = \varnothing, \tag{4.46}$$

where $\hat{P}_{b-1}^{l-1}$ is the set of the expanded predicates with arity $b-1$ at column $l-1$, and $\check{P}_{b+1}^{l-1}$ is the set of the reduced predicates with arity $b+1$ at column $l-1$.

The language used by DLM is more expressive than Datalog, and DLM requires only a few hyper-parameters to be trained: L, B, n_O, and n_A. The architecture of DLM allows to adopt these parameters with fine granularity. For example, the user can specify different n_O values for different modules. Unfortunately, DLM has the same problem of deep neural networks, which are difficult to design for a specific task.

For ILP tasks, DLM computes binary cross-entropy between the predicted label and the actual one. At the end of the training, DLM offers the possibility to extract the logical

ground rules, replacing Gumbel-softmax with argmax and fuzzy operators with boolean operators. The algorithm starts from the output of the network and moves backward to extract the output rules, which can be used to complete the task more efficiently.

From a performance perspective, the results that are reported in the original paper [10] suggest that DLM is generally comparable to δILP and NLM, and it is slightly better than dNL-ILP. DLM manages to perform equal or better than the other methods in all the proposed tasks, and it does not suffer from the memory-related problems that are reported for δILP and dNL-ILP. In general, DLM is an interesting method, and it allows to obtain interpretable solutions for ILP problems using only a few hyper-parameters. Moreover, it is very expressive, and it allows inducing complex FOL rules for a given task. However, the language employed for the resulting formulae does not offer the interesting features of Datalog, and the method is able to induce ground rules only. Ground rules are not always reusable in tasks different from the one used for training, and this prevents the use of DLM in many real-world applications. Moreover, the performance of DLM is reported only for a few tasks, and more tests are needed to understand the real potential of this method.

4.2.6 Meta$_{\mathrm{Abd}}$

Previous methods, like traditional ILP algorithms, assume that the input of the learning task is symbolic. Therefore, these algorithms assume the availability of a coupled algorithm to map non-logical data to their logical counterparts before the learning process. On the contrary, Meta$_{\mathrm{Abd}}$ [22] follows a different approach, and it tries to map sub-symbolic input, like images, to symbolic one and, at the same time, it induces a set of rules that defines the target predicates.

Meta$_{\mathrm{Abd}}$ is composed of two cascaded modules: a perception module and a reasoning module. The perception module performs the mapping between sub-symbolic input x and symbolic interpretation z. For example, if x is an image of an handwritten digit, z is the corresponding integer. The perception module is a neural network with parameters θ that estimates the conditional probability $P(z|x, \theta)$. The other module is the reasoner H, which is composed of a set of rules that can be used to infer an output symbol y, provided that background knowledge B and z are available. The goal of Meta$_{\mathrm{Abd}}$ is to learn θ and H simultaneously from training data.

Due to the large search space of interesting ILP tasks, it is very difficult to learn θ and H simultaneously. Therefore, to overcome this problem, Meta$_{\mathrm{Abd}}$ makes use of a combination of induction and abduction. Abductive reasoning (e.g., [64]) is a form of logical inference in which the goal is to infer facts and rules that, using background

knowledge, provide the best explanation of training examples. Meta_{Abd} induces H in terms of an abductive theory expressed in terms of abducible primitives, which actually form the background knowledge. The hypothesis H is then used to prune the search space for z by abducing facts that constraint the output of the perception module, which computes $P(z|x, \theta)$, to guide the search for the most probable combination of H and z.

If D is the set of training samples, Meta_{Abd} learns from training examples $\langle x_i, y_i \rangle \in D$ using the perception module and the background knowledge B that consists of abducible primitives. In order to perform the training, Meta_{Abd} employs the following objective function:

$$\theta^* = \underset{\theta}{\text{argmax}} \prod_{\langle x_i, y_i \rangle \in D} \sum_{H \in \mathcal{H}} \sum_{z \in \mathcal{Z}} P(H, z|B, x, y, \theta) \tag{4.47}$$

where $\mathcal{H}$ is the hypothesis space and $\mathcal{Z}$ is the interpretation space. Meta_{Abd} uses an expectation maximization algorithm to learn H using two steps:

1. Expectation: $P(H, z|B, x, y, \theta)$ is used to sample the expected values of H and z; and

2. Maximization: new parameters θ are estimated using a stochastic gradient descent algorithm.

In the expectation step, Meta_{Abd} induces H, which is then used to abduce z. The algorithm assigns a score to each pair $H \uplus z$, and then it chooses the one with the highest score. In particular, the expectation step is structured in four steps:

1. An abductive theory $H \sim P_{\sigma^*}(H|B)$ is induced. $P_{\sigma^*}(H|B)$ is the Bayesian prior distribution of first-order logic hypothesis, and it is computed using the operator σ^* [65];

2. Meta_{Abd} uses the hypothesis $H \cup B$ and y to abduce the possible values for z. These values are guaranteed to satisfy $(H \cup B) \uplus z \models y$ and $P(y|B, H, z) > 0$;

3. The algorithm assigns a score to $H \uplus z$:

$$score(H, z) = P_{\sigma^*}(H|B)P_{\theta}(z|x); \tag{4.48}$$

4. Finally, Meta_{Abd} selects the $H \uplus z$ with the higher score.

In order to induce new rules, Meta_{Abd} uses a form of higher-order logical meta-rules, which can be written as:

$$metarule([P, Q], [P, A, B], [[Q, A, B]]), \tag{4.49}$$

where the above meta-rule represents the direct application of an abducible primitive $P(A, B) \Leftarrow Q(A, B)$. The meta-rules of Meta_{Abd} are similar to those used by other neural-symbolic approaches for ILP (e.g., [16,21]) because they all define a general structure of the generated rules that is applicable to different predicate symbols. Meta-rules represent an interesting way to implement recursion and predicate invention, but they are limited because induced rules must follow a predetermined structure that is defined by the human expert. In many applications, the requirement of the a priori understanding of the rule structure is not practical, and such a requirement may prevent the use of Meta_{Abd} and similar approaches in complex problems. In the original paper [22], authors do not directly compare their method with the other methods discussed in this chapter. Moreover, it is not clear how well Meta_{Abd} scales with the size of the training set. Finally, it would be interesting to measure the performance of Meta_{Abd} on tasks that go beyond simple toy problems.

4.3 Comparing methods from the XAI perspective

This section presents a comparison of neural-symbolic approaches for ILP paying particular attention to the interpretability of the generated rules. Discussed approaches are all based on templates or meta-rules. This section is an extension of the comparison proposed in [23], and it follows a classification that is similar to the classification used in [53]. The considered methods are δILP, NTPs, ILP_{Camp}, dNL-ILP, and Meta_{Abd}. DLM has not been considered in this comparison because it produces ground rules only. Therefore, it would be unfair to compare DLM with the other methods because they produce different types of rules. The compared methods are categorized using four characteristics: the language used to express the learned rules, the adopted search method, the possibility to learn recursive rules, and the possibility to learn invented predicates. This classification is useful to analyze the interpretability of the learned rules. In fact, the used language as well as the support of predicate invention and recursive rules are characteristics that have a strong impact on the interpretability of the learned rules. At the same time, the search method influences how the rules are induced. Therefore, it has been included in this classification because it can directly influence the structure of the learned rules. Table 4.1 summarizes the comparison of the four methods, which is detailed in the rest of this section.

4.3.1 Language

The language used to represent the learned rules is an important characteristic of each method, especially from the point of view of XAI [50]. Although there are several methods that do not support FOL, like, for example, NLM [62], all the discussed methods produce programs in a Datalog dialect. In particular, δILP learns programs written in Datalog, while Meta_{Abd} learns programs written in a specific dialect of Datalog. Both methods produce explicit rules in their respective languages. Similarly, dNL-ILP produces rules in DNF or CNF. In particular, a rule written in DNF can be used to define a predicate, and it is equivalent to defining a set of Datalog rules. For example, lets consider the definition of a less-than relation over natural numbers. As shown in [19], such relation could be represented using the predicate `lessThan`, defined as follows:

```
lessThan(A,B) :- inc(A,B),
lessThan(A,B) :- lessThan(A,C),inc(C,B),
```

where `inc` represent an increment of 1. The predicate `lessThan` can also be written as a DNF rule:

$$\texttt{inc(A,B)}\vee(\texttt{lessThan(A,C)}\wedge\texttt{inc(C,B)}).$$

Unlike the previously discussed methods, NTPs assume the existence of a Datalog knowledge base, and they learn the real vectors associated with rules. The generated rules can then be used to deduce new ground atoms. In NTPs, rules are not expressed as explicit Datalog programs, but they are used in the context of knowledge base completion. Finally, from an XAI perspective, ILP_{Camp} and NTPs share the common problem of encoding rules using real vectors, which is a limitation for the interpretability and the explainability of the learned rules. In principle, it is possible to extract the explicit relationship among the vectors associated with learned predicates by measuring their similarity, but this possibility is not currently included in these methods.

4.3.2 Search method

The most common approaches to ILP are classified into top-down and bottom-up. Top-down methods start with a general hypothesis that is gradually specialized. Bottom-up methods start from training data, and they produce a specialized hypothesis that is gradually generalized. Recently, meta-level approaches have been introduced to encode the ILP task as a meta-level program, which is a program that reasons on programs. This meta-level program is then used to compute the proposed solutions, which are then

Table 4.1: Summary of the features that characterize surveyed methods.

Method	Language	Search method	Recursion	Predicate invention
δILP	Datalog	Meta-level	Yes	Partially
NTPs	Datalog	Top-down	Yes	No
ILP_{Camp}	Datalog	Top-down	Yes	Partially
dNL-ILP	Datalog	Meta-level	Yes	Partially
Meta_{Abd}	Datalog	Meta-level	Yes	Yes

translated back to inductive solutions of the ILP task [53]. Among the five presented methods, NTPs and ILP_{Camp} follow a top-down approach, while δILP, dNL-ILP and Meta_{Abd} follow a meta-level approach.

Actually, δILP generates different logical hypotheses using the restrictions imposed by the program template. δILP starts with no assumptions on the final solution and, at each training step, the weights associated with the generated rules are updated. δILP makes use of a differentiable framework as a meta-level program to find the most appropriate solution to the ILP task. Similarly to δILP, dNL-ILP requires the user to specify a program template, and it uses a deep neural network to select the most appropriate set of literals that define a predicate. NTPs and ILP_{Camp} are top-down methods because the structure of the induced rules are specified by fixed meta-rules. Meta-rules describe the most general solution to the ILP tasks, and they are specialized by training the weights associated with the abstract predicate symbols. Meta_{Abd} is an extension of *Meta-Interpretive Learning* (*MIL*) [56], which is a meta-level method. In particular, Meta_{Abd} makes use of higher-order meta-rules to generate hypotheses, and it uses a modified MIL Prolog meta-interpreter to search the space of hypotheses for a solution of the ILP task.

4.3.3 Recursion

Recursion is an important feature in ILP because it allows performing an infinite number of deduction steps with a finite logic program. The support for recursion makes ILP methods able to better generalize from small numbers of examples because the methods do not need to learn a separate rule for each specific situation. All mature methods to accomplish ILP tasks are expected to support recursion, and all the discussed methods support recursion. In particular, δILP and dNL-ILP provide recursion by design because

they use program templates to generate rules that include recursive rules. Recursion is enforced in NTPs by manually specifying meta-rules that describe how recursion is expected to be used in generated rules. The following is an example of a meta-rule that can be used to learn recursive rules using NTPs:

$$2\ \#m_1(X,Y) \text{ :- } \#m_1(Y,X) \tag{4.50}$$

which is instantiated only two times, and whose predicates, in the head and in the body, are the same. Meta_{Abd} and ILP_{Camp} use a similar form of meta-rules, but they do not expect the user to manually specify the relationships among the rule predicates. Instead, they represent predicates with a generic predicate, and the methods are expected to learn to correctly associate each predicate in the meta-rule with the corresponding predicate defined as part of the ILP task.

4.3.4 Predicate invention

The induction of rules from training data is an extremely difficult task, and the choice of an appropriate background knowledge is crucial to improve the learning performance. Background knowledge is typically provided by experts as a hand-crafted set of facts and rules, but in most real-world applications, it is difficult, and often impossible, to provide a comprehensive knowledge base. Predicate invention is intended to automatically generate new predicates, under suitable constraints, obviating the user from manually specifying the knowledge needed to solve the ILP task.

The support for predicate invention is considered a major challenge, and most ILP methods do not provide this feature. δILP and dNL-ILP support predicate invention only partially. Actually, they require the user to manually specify which auxiliary predicates to learn—predicate symbol and arity—in order to define the target predicate. As discussed previously in this section, it is often very difficult to foresee the number and the arity of auxiliary predicates, which makes δILP and dNL-ILP unsuitable for most real-world applications. ILP_{Camp} too supports predicate invention only partially, and it only allows the user to specify the number of auxiliary predicates [21]. The method then learns their representations. As δILP, ILP_{Camp} requires the user to know in advance some characteristics of the final solution, although ILP_{Camp} provides a more flexible approach because it does not need to know the arity of auxiliary predicates. NTPs do not support predicate invention, but it is expected that predicate invention could be included in NTPs because NTPs and ILP_{Camp} share a similar approach to rule induction. Meta_{Abd} is an extension of the MIL interpreter, which natively supports predicate invention. In particular, Meta_{Abd} is able to use meta-rules to reduce the space of hypotheses and to

Table 4.2: Summary of the features that characterize surveyed algorithms, where the representation of data is detailed for the background knowledge (BK column) and for the training set (Dataset column).

Algorithm	BK	Noisy BK	Dataset	Language bias
δILP	Facts	Yes	Facts	Temp. & const.
dNL-ILP	Facts	Yes	Facts	Template
Meta_{Abd}	Rules	No	Images	Meta-rules

invent new predicates whenever necessary. Invented predicates are then defined using the same meta-rules provided as part of the description of the ILP task.

4.4 Comparing Datalog-based methods

This section presents a comparison of three promising neural-symbolic approaches for ILP, namely δILP, dNL-ILP, and Meta_{Abd}, paying particular attention to the reusability of the learned rules. The discussed approaches generate logical rules in Datalog dialects, so that learned rules can be used to solve the particular learning problem and to perform logical deduction to extend the background knowledge. DLM has not been considered in this comparison because it does not produce Datalog rules, while NTPs and ILP_{Camp} are designed to extend a knowledge base. Therefore, they have not been considered in this comparison because the induction of Datalog rules is not their primary goal. Actually, many neural-symbolic approaches for ILP do not produce reusable rules or they use specific subsets of first order logic that are normally less powerful than Datalog because they are designed for the specific task at hand. The discussed algorithms are compared using two characteristics: the representation of the data used for the training examples and the background knowledge, and the language bias that is enforced to guide the generation of rules. This section is a summary of the comparison proposed in [24], but the support of recursion and predicate invention are not described here because they have been already discussed in the previous section. Table 4.2 summarizes the proposed comparison among the three studied algorithms using these characteristics, as detailed in the remaining of this section. Note that the table details the representation of data used for the training examples (Dataset column), and the representation of data used for the background knowledge (BK column).

4.4.1 Representation of data

The representations of background knowledge and training examples plays an important role in the considered learning tasks. Normally, ILP algorithms assume that the background knowledge is composed of a set of facts and logical rules, while the training set is a set of facts that are used to define the target predicate. Unfortunately, this representation of the background knowledge and of the training set cannot adequately treat noisy and erroneous data, and it must be reconsidered in the scope of neural-symbolic approaches for ILP.

As far as the three studied algorithms are concerned, training examples are provided in different forms: both δILP and dNL-ILP define the target predicate using a set of facts, while Meta_{Abd} requires the training set to consist of images. The adoption of images for the training set represents a clear advantage when the learning task is required to work on raw pixels. It is worth noting that δILP was coupled with a perception module implemented using an artificial neural network trained to recognize handwritten numbers. δILP was then tested together with this perception module, but the results was not considered satisfactory [15]. This suggests that a much finer integration between a perception module and a neural-symbolic algorithm–like the integration proposed by Meta_{Abd}–can be used to obtain considerable improvements. As a side note, it is worth mentioning that all the studied algorithms allow mislabelled examples because they all try to minimize a loss function rather than satisfying a strict constraint, which is what traditional ILP algorithms do.

In the case of background knowledge, instead, both δILP and dNL-ILP define the initial predicates as a set of facts, while Meta_{Abd} defines the background knowledge as a set of clauses. This represents an advantage for Meta_{Abd} because it does not require to manually specify the set of ground atoms in order to define the initial predicates. In particular, the choice of clauses allows handling infinite domains because the description of background knowledge as a set of facts makes the learning impractical as the number of domain elements grows. From the perspective of noisy background knowledge, both δILP and dNL-ILP assign a value in $[0, 1]$ to background facts, while Meta_{Abd} expects that B is exact and free from uncertainty. This could represent a problem for Meta_{Abd} in some applications, because background knowledge is often imperfect and uncertain.

4.4.2 Language bias

As said, the induction of rules requires searching in a large search space. In order to guide the search, ILP algorithms normally employ a language bias, which is typically

defined as a set of rules or constraints that partially define accepted rules. δILP defines different restrictions for the generation of rules. As mentioned, a predicate can be defined by only two rules in δILP, but other constraints are imposed on the generation of rules, as discussed previously. These restrictions on the generation of rules combined with a program template allow δILP to effectively learn from data, although the approach is not sufficiently scalable because the number of weights grows quickly as the difficulty of the task increases. On the other hand, dNL-ILP is more scalable, and it requires only some parameters to define the search space for the generation of rules. Rules are generated by only specifying the definition of the target and of the auxiliary predicates, namely, the name and the arity of each predicate together with the number of variables in the body of each rule. Although these two algorithms use some form of template–as a specific set of parameters–to restrict the search space, dNL-ILP represents a more flexible approach than δILP because the number of required weights is smaller and the structure of the induced rules is less constrained. Meta_{Abd} follows another approach, and it employs meta-rules to define the structure of the generated rules. Using this approach, the human expert can use the domain knowledge to improve the learning performance, but it is rarely possible to accurately foresee the structure of the solution to a particular task, and this makes meta-rules impractical for real-world applications.

4.5 Open challenges

Although there is a growing interest on neural-symbolic methods for ILP, several problems remain to be tackled. From the perspective of the supported features, many approaches do not fully support automatic predicate invention. Moreover, the language biases used to guide the generation of rules impose too many constraints, or they require the human expert to inject too much domain knowledge to effectively learn an appropriate solution to a learning task. Both meta-rules and program templates are not suitable for real-world tasks, and an interesting attempt to target these problems is DLM [10], which uses a more complex subset of FOL than Datalog. Unfortunately, DLM produces ground rules only, but it can be considered as an interesting starting point to overcome the mentioned problems. In addition, only a few algorithms are capable of effectively learning program written in a Datalog dialect from noisy data.

From the user perspective, neural-symbolic ILP methods are often difficult to use because they lack a complete and user-friendly implementation. When available, these implementations are typically not well documented and maintained. Therefore, it would be necessary to pay more attention to these aspects to reach a wider audience and to

increase the research effort on this subject. In addition, it is difficult to perform a comparison of neural-symbolic ILP methods from the perspectives of performance and scalability because there is no standard set of benchmarks for ILP algorithms. Paying more attention to these aspects is important to reach a wider audience and to increase the interest of researchers in ILP and related challenges. An interesting development could be the design and the implementation of a tool like OpenAI Gym [27], which provides a benchmark suite for RL algorithms, to specifically assess ILP methods. Unfortunately, Gym does not provide a logical interface to the RL tasks, which could be used to train neural-symbolic methods. A tool like OpenAI Gym could provide different learning problems, from toy problems to real-world problems, allowing researchers to effectively measure the characteristics of proposed methods in different situations.

Other aspects to consider are related to the representation of data and to the treatment of noisy background knowledge, as showed in the second comparison. Meta_{Abd} defines background knowledge as a set of rules. On the contrary, the other studied algorithms require the knowledge base to contain only ground atoms. Therefore, Meta_{Abd} is capable of handling infinite domains, unlike δILP and dNL-ILP. On the other hand, Meta_{Abd} does not support noisy background knowledge, which is instead supported by the two other methods. None of the three approaches support both these features, and it would be interesting to further investigate the possibility to support both these features. Finally, in most ILP algorithms, the training set is composed of logical facts and rules. Meta_{Abd} is one of the few algorithms that natively supports images as training examples. It would be interesting to further investigate in this direction, designing a method that natively learns logical rules from complex images.

Symbolic techniques have several advantages over sub-symbolic techniques, including the possibility of extending an existing knowledge base for transfer or lifelong learning. In order to avoid the explosion of the background knowledge, some form of optimization among rules is necessary. Many induced rules may represent specific cases, even if a more general rule could be already available. The reduction of the complexity of rules, and the removal of unnecessary rules, represents a major challenge to be addressed in the future.

Finally, it is worth noting that, especially when using predicate invention, the learned rules may be difficult to interpret. The optimization of rules could help improve the interpretability of the learned rules, but a greater effort should be made to guarantee that the generated rules are understandable by humans.

From the comparisons emerges that several improvements has been made, with features like recursion and predicate invention that are more and more supported. However, many problems remain to be tackled. Nonetheless, the works discussed in this disserta-

tion witness that neural-symbolic ILP has the potential to overcome the limitations of traditional ILP, allowing to produce reusable and interpretable solutions to real-world problems. As already mentioned, ILP methods could represent a good starting point to deal with RL tasks. The next chapter presents a new neural-symbolic approach to RL tasks which is indirectly based on δILP.

Chapter 5

The Proposed Method SD-NLRL

This chapter presents a new neural-symbolic method for *Reinforcement Learning* (*RL*). The RL literature presents only a few neural-symbolic methods. *Neural Logic Reinforcement Learning* (*NLRL*) [9], DLM [10], and dNL [11], which is an adaptation of dNL-ILP [19,20] for RL, are interesting examples of neural-symbolic methods. However, the existing neural-symbolic methods for RL requires the user to specify a large amount of information to effectively solve a RL task. This represents an important limitation because it is often very difficult, or even impossible, to provide such information for real-world problems. The proposed method is based on NLRL, which is an adaptation of δILP for RL. Here, for the sake of clarity, the proposed method is called *State Driven NLRL* (*SD-NLRL*).

This chapter is structured as follows. Section 1 discusses the motivations behind the choice of NLRL as the base of SD-NLRL. Section 2 presents the NLRL algorithm, describing the main differences with the original ILP method, namely, δILP. Section 3 describes the five tasks that are used to test the performance of the algorithms: ON, STACK, UNSTACK, CLIFFWALKING, and WINDYCLIFFWALKING. Section 4 introduces SD-NLRL, discussing the differences with NLRL and the proposals that have been explored for this work. In particular, four versions of SD-NLRL are discussed, and only the last version reaches a reasonable level of performance. The other three variants represent the important steps that have been made to obtain the final version of SD-NLRL. Experimental results are presented for twp proposed modification. Finally, Section 5 concludes this chapter discussing the main problems to be solved and some interesting ideas that can be explored in the future.

5.1 Motivation

NLRL has been chosen as the base of SD-NLRL for three main reasons:

1. NLRL produces explicit Datalog rules with reasonable performance;
2. The architecture of NLRL can be easily adapted to produce state-driven rules; and
3. The implementation of NLRL is available and easy to inspect.

Other two RL methods that were taken into consideration are: DLM and dNL. From the perspective of the language used for learned rules, DLM produces ground rules, which makes this method unsuitable in many real-world applications. Moreover, the support of state-driven rules is important because the goal of the proposed method is to free the user from specifying a large number of domain-specific parameters, while, at the same time, producing interpretable policies. Compared to dNL, NLRL works on the level of rules, defining a weight for each definite clause. The generation of rules from the states of the environment, which is the goal of SD-NLRL, is straightforward using the NLRL architecture. Finally, the availability of an inspectable implementation is also important because having a usable implementation allows performing an accurate comparison between the original algorithm and the modified one. In fact, most neural-symbolic RL methods do not provide an usable implementation. Moreover, when the implementation is available, a standard set of tasks that can be used to test the performance of the algorithms is not available. In particular, DLM offers an usable implementation, but it outputs the generated rules in an non-readable format. dNL offers a working implementation that is conceived for ILP tasks only. Therefore, in order to test the ideas that are the foundations of this work, NLRL has been chosen as a base for the SD-NLRL implementation.

5.2 NLRL

NLRL is an adaptation of δILP [15], for RL tasks. The paper that introduces NLRL does not discuss many important details on the actual implementation, the information that is not explicitly available in the original paper is taken from the available implementation[1]. NLRL requires the user to provide a background knowledge expressed using Datalog rules, similarly to δILP. Moreover, the environment is also described as a set of ground atoms, and each ground atom indicates whether a specific characteristic of

[1] `https://github.com/ZhengyaoJiang/NLRL`

the environment is present. For example, a state of the popular game Tic-Tac-Toe can be represented as the set `[first(1,2),second(3,3),first(2,2),second(3,2)]`, where `first(K,Y)` and `second(K,Y)` are two predicates that represent X and O marks on the table at position (K,Y), respectively. Finally, the agent interacts with the environment taking actions, which are defined by action predicates, and each action predicate must be defined as part of the environment. In particular, NLRL tries to learn the best set of rules that define each action predicate. For example, the actions for Tic-Tac-Toe can be defined as `[drawFirst(1,2),drawSecond(3,4),...]`, where `drawFirst(K,Y)` and `drawSecond(K,Y)` are two action predicates that draw X and O marks on the table at position (K,Y), respectively.

Let G be the set of ground atoms defined for the task. In order to tackle RL problems, NLRL introduces the concept of *MDP with Logic Interpretation* (*MDPLI*). An MDPLI is defined as a tuple $\langle M, p_S, p_A \rangle$, where:

- M is a finite-horizon MDP;
- $p_S : S \to 2^G$ is a function that performs the encoding of a state, which is transformed in a set of ground atoms;
- $p_A : [0,1]^{|D|} \to [0,1]^{|A|}$ is a function that maps a valuation of a set of atoms D to a vector representing the probabilities of each action.

The output of p_S includes information on both the current state and the background knowledge. The learning method is a modified version of the δILP algorithm, and it is formally defined as function $f_\theta : 2^G \to [0,1]^{|D|}$. The function f_θ takes as argument a set of ground atoms representing the current state, and it produces a valuation vector representing the assigned score to a set of D ground atoms. The policy can be defined as:

$$\pi(s) = p_A(f_\theta(p_S(s))) \tag{5.1}$$

for all $s \in \mathcal{S}$. In general, any Policy Gradient method that can solve *Deep Reinforcement Learning* (*DRL*) problems can be used to implement f_θ. The original paper makes use of REINFORCE [34]. Both p_A and p_S can be implemented using hand-crafted procedures or artificial neural networks. The output of f_θ is defined for a generic set of ground atoms D because it is typically difficult to produce a probability value using the action atoms only. The original paper discusses a simplified version of NLRL where $D = G$. Therefore, the output of the learning algorithm is a valuation vector containing a score for each ground atom. In particular, p_A and p_S are defined as ad-hoc functions: p_S simply transforms the states into their logical representations, and p_A computes the probability

of the actions using the scores produced by f_θ. It is worth noting that both p_A and p_S can be implemented using artificial neural networks. Therefore, p_S can easily work on images or other non-logical input to build a logical representation of the state, and p_A can implement a more flexible and sophisticated strategy to select actions.

The architecture of NLRL requires p_A to be a differentiable function. In fact, making p_A differentiable allows the gradients to pass from the output back to the weights of the rules. Now, the hand-crafted function p_A that is proposed by the authors of NLRL is formally defined. Let $a \in A$ be an action atom, and $e \in [0,1]^{|D|}$ be a valuation vector. Then, p_A can be defined as:

$$p_A(e) = \{P(a|e) \mid \forall a \in A\} \quad \text{where} \quad P(a|e) = \begin{cases} \frac{l(e,a)}{\sigma} & \text{if } \sigma \geq 1 \\ l(e,a) + \frac{\sigma}{|A|} & \text{otherwise} \end{cases} \tag{5.2}$$

for all $e \in [0,1]^{|D|}$. The function $l : [0,1]^{|D|} \times A \to [0,1]$ extracts the valuation of a single atom from a valuation vector, and σ is the sum of the action valuations: $\sigma = \sum_a P(a|e)$. Using 5.2, the probability of choosing action a is proportional to its score only if the sum of all action valuations is equal or greater than 1. Otherwise, NLRL evenly distributes to all action atoms the difference between 1 and the sum of all scores. This design of p_A is motivated to increase the differences between the weights of desired actions and the weights of undesired actions. Authors empirically report that defining p_A as in 5.2 can produce more interpretable and generalizable policies.

The core of NLRL, represented by f_θ, is defined using the δILP method, as presented here recalling the concepts that are discussed in 4.1.2 for completeness. The core function $f_\theta : 2^G \to [0,1]^{|G|}$ takes a set of ground atoms G as argument, and it tries to induce Datalog rules that are used to assign a probability to each action predicate. In particular, f_θ takes a set of ground atoms B as argument that represents the background knowledge. Then, f_θ generates all the possible rules that can be used to implement the action predicates. As discussed previously, the induction of definite clauses from data requires searching in a large search space that is dependent on the size of B. In order to prune the search space, f_θ makes use of a program template, defined as follows:

$$\Pi = \langle \text{Pred}_a, \text{arity}_a, (\tau_p^1, \dots, \tau_p^k), T \rangle, \tag{5.3}$$

where Pred_a and arity_a are the sets of auxiliary intensional predicates and of their arities, respectively. $T \in \mathbb{N}$ is the number of forward chaining steps, and τ_p^j are rule templates that specify how f_θ generates each rule for each intensional predicate $p \in \text{Pred}_i$. Each rule template is defined as:

$$\tau = \langle v, int \rangle, \tag{5.4}$$

where $v \in \mathbb{N}$ represents the number of free variables in the rule, and $int \in \{0, 1\}$ indicates if the intensional predicates are allowed in the rule. It is worth noting that B and Π must be specified by the user, while the other information is defined as part of the task. A major difference between NLRL and δILP is that NLRL allows a target predicate to be defined by more than two rules. For each rule template τ, f_θ generates the set of rules $cl(\tau)$ enforcing the same restrictions that δILP enforces:

1. Constants are not allowed in $cl(\tau)$;
2. Predicates of arity higher than three are not allowed in $cl(\tau)$;
3. Each rule in $cl(\tau)$ must contain exactly two atoms in its body;
4. All variables that appear in the head of a rule in $cl(\tau)$ must appear also in its body;
5. Two rules that differ only in the order of the atoms in their bodies are not both allowed in $cl(\tau)$; and
6. An atom is not allowed at the same time in the head and in the body of a rule in $cl(\tau)$.

Let cl_p be the set of rules that are generated for predicate $p \in \text{Pred}_i$. cl_p is composed of the rules that are generated using rule templates τ_p^j. The size of cl_p is defined as $|cl_p| = \sum_{j=1}^{k} |cl(\tau_p^j)|$. For each predicate p, NLRL defines a vector of weights $W_p \in \mathbb{R}^{|cl(\tau_p^1)| \times \dots \times |cl(\tau_p^k)|}$, and each weight is associated with a rule in cl_p. A softmax function is applied to W_p in order to obtain a probability distribution $W_p^* \in [0, 1]^{|cl(\tau_p^1)| \times \dots \times |cl(\tau_p^k)|}$:

$$W_{p,k}^*[j] = \frac{e^{W_{p,k}[j]}}{\sum_{j'} e^{W_{p,k}[j']}}. \tag{5.5}$$

where $W_{p,k}^*[j]$ denotes the probability that rule j is the best rule to define the k-th rule of predicate p. The set of vectors W_p* is denoted as W.

As δILP, NLRL uses a continuous relaxation of the truth value associated with a ground atom. Let G be a set of n ground atoms. Recall that a valuation e is a vector in $[0, 1]^n$, where each element $e_i \in [0, 1]$ is the fuzzy truth value of atom $\gamma_i \in G$. NLRL does not define a training set Λ like δILP does. In particular, NLRL takes, for each rule, T forward chaining steps, and it tries to predict the correct probability value for each action predicate. In a similar way to δILP, NLRL makes use of background knowledge B and language L (consisting of constants, action predicates, and extensional predicates), and it tries to predict the following probability:

$$P(\gamma | W, \Pi, L, B) = f_{\text{extract}}(f_{\text{infer}}(f_{\text{convert}}(B), f_{\text{generate}}(\Pi, L), W, T), \gamma), \tag{5.6}$$

for each action predicate $\gamma \in G$. The computation of $P(\gamma|W, \Pi, L, B)$ is very similar to the one that has been presented for δILP:

1. Function $f_{\text{convert}} : 2^G \to [0, 1]$ is applied to retrieve the initial truth value of each ground atom:

$$f_{\text{convert}}(B) = y \quad \text{where} \quad y[i] = \begin{cases} 1 & \text{if } \gamma_i \in \text{B} \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

   and $\gamma_i \in G$ is the i-th ground atom.

2. Then, the set of possible rules is generated using f_{generate}:

$$f_{\text{generate}}(\Pi, L) = \{cl_p |\ p \in \text{Pred}_i\}. \tag{5.8}$$

3. Then, using function $f_{\text{infer}} : [0, 1]^n \times C \times W \times \mathbb{N} \to [0, 1]^n$, NLRL takes T forward chaining steps to obtain the predicted truth value for each action predicate.

4. Finally, function $f_{\text{extract}} : [0, 1]^n \times G \to [0, 1]$ is used to retrieve the probability of an action atom γ:

$$f_{\text{extract}}(x, \gamma) = x[index[\gamma]], \tag{5.9}$$

   where $index : G \to \mathbb{N}$ computes the unique index of a ground atom.

Note that 1 and 2 are executed only at the beginning of the training phase. Then, NLRL continuously interact with the environment to collect complete episodes. Each episode is subdivided in batches of b subsequent samples. For each sample, function p_S is applied to retrieve the truth values of ground atoms G that represent the current state of the environment. Then, 3 and 4 are used to compute the predicted truth value of each action atom. Finally, p_A is applied to compute the probability of each action.

In order to train the weights W, NLRL interacts with the environment, and it computes an appropriate loss function. In particular, an action is chosen using the probability distribution on action atoms for each sample. The loss function is defined as the log-likelihood of the chosen actions:

$$loss = -\sum_{i=1}^{b} log(P(a_i))Adv(s_i, a_i) \tag{5.10}$$

where a_i is the selected action at time step i, and $Adv(s_i, a_i)$ is the advantage function of state s_i and action a_i. NLRL tries to minimize the loss function using an optimizer, enforcing actions a_i that are better than the other actions in the current state s_i. The

advantage function *Adv* is implemented using an artificial neural network that is trained together with weights W. In particular, the sequence of states and rewards that defines the current batch of samples is used to train a feed-forward neural network that tries to predict the return of a given state. The advantage is then computed using *Generalized Advantage Estimation* (*GAE*) [66]. Let $t \in \{1, \ldots, b\}$ be the index of a sample in the current batch, $GAE(\gamma, \lambda)$ is defined as:

$$A_t^{GAE(\gamma,\lambda)} = \sum_{j=1}^{\infty} (\gamma\lambda)^j \delta_{t+j} \quad \text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5.11}$$

and $\gamma \in [0, 1]$ is the discount rate, and $\lambda \in [0, 1]$ is another parameter that is used to adjust the trade-off between bias and variance. Value function V is estimated using the discussed neural network, and the NLRL implementation modifies the original GAE algorithm by using the reward of the last sample as the corresponding advantage, i.e., $A_b^{GAE(\gamma,\lambda)} = r_b$.

Function f_{infer} is defined in a similar way to the one that has been presented for δILP, and it is based on functions $F_c : [0, 1]^n \to [0, 1]^n$, which are defined for each rule c. Each function F_c takes the current truth value of the ground atoms as a valuation, and it computes a new valuation that represents the truth value of the ground atoms after the application of rule c. Like δILP, NLRL defines a mapping between each ground atom and a unique index, and it prepares a matrix $X_c \in \mathbb{N}^{n \times w \times 2}$ before starting the training phase, where w is the maximum number of index pairs relative to ground atoms that logically entail each ground atom. X_c is formally defined as:

$$\begin{gathered} x_k = \{(a, b) \mid \text{satisfies}_c(\gamma_a, \gamma_b) \wedge head_c(\gamma_a, \gamma_b) = \gamma_k\} \\ X_c[k, m] = \begin{cases} x_k[m] & \text{if } m < |x_k| \\ (0, 0) & \text{otherwise} \end{cases} \end{gathered} \tag{5.12}$$

where x_k is a set of index pairs, and each pair references two ground atoms that entail the k-th ground atom. Similarly to δILP, in NLRL X_c is not fully used because the maximum number of index pairs w is used to build the structure of X_c. Therefore, NLRL, as well as δILP, includes a falsum atom $\perp$ in G, and each unused pair of indexes $(0, 0)$ is mapped to the atom pair $(\perp, \perp)$.

During the training phase, NLRL takes two slices of X_c, namely, $X_1, X_2 \in \mathbb{N}^{n \times w}$. X_1 and X_2 represent the first and second element of each pair, respectively:

$$X_1 = X_c[_, _, 0] \quad X_2 = X_c[_, _, 1] \tag{5.13}$$

Then, the algorithm builds two matrices $Y_1, Y_2 \in [0, 1]^{n \times w}$ using the function $gather_2$:

$\mathbb{R}^a \times \mathbb{N}^{b\times c} \rightarrow \mathbb{R}^{b\times c}$:

$$gather_2(x,y)[i,j] = x[y[i,j]] \tag{5.14}$$

$$Y_1 = gather_2(a, X_1) \quad Y_2 = gather_2(a, X_2) \tag{5.15}$$

Y_1 and Y_2 represent the current fuzzy truth values of a valuation a that are referenced by X_1,X_2. NLRL uses Y_1 and Y_2 to define a matrix $Z_c \in [0,1]^{n\times w}$ as:

$$Z_c = Y_1 \odot Y_2 \tag{5.16}$$

Element-wise multiplication $\odot$ is used to implement a fuzzy conjunction between the truth values of the ground atoms that form the body of rule c. Finally, NLRL defines F_c as:

$$F_c(a) = a' \quad \text{where} \quad a'[k] = \max(Z[k, _]) \tag{5.17}$$

Here, for each ground atom γ_k, taking the maximum value implements a disjunction between the truth values of the fuzzy conjunctions that produce a truth value for γ_k.

The second major difference between δILP and NLRL is the sequence of operations needed to compute a forward chaining step. In particular, δILP defines a forward chaining step for each pair of rules using the max operator, while NLRL combines the truth values of each rule using a probabilistic sum operator. Recall that the probabilistic sum is defined as:

$$a \oplus b = a + b + a \cdot b \quad \text{where} \quad a, b \in [0,1] \tag{5.18}$$

NLRL uses f_{convert} to obtain an initial valuation a_0:

$$a_0[x] = \begin{cases} 1 & \text{if } \gamma_x \in \text{B} \\ 0 & \text{otherwise} \end{cases} \tag{5.19}$$

Then, it defines a forward chaining step as:

$$a_{t+1} = a_0 + \left(\sum_{p\in\text{Pred}_i} \sum_{k}^{\oplus} \sum_{j} W^*_{p,k}[j] F^j_{p,k}(a_t) \right) \tag{5.20}$$

where $W^*_{p,k}[j]$ is the weight associated to the j-th rule generated from the k-th program template for predicate p. Similarly, $F^j_{p,k}$ implements a deduction step for the j-th rule generated from the k-th template for predicate p. The output of each step a_{t+1} is dependent only on the initial valuation a_0 and on the deduced valuation a_t. Therefore, definitions of predicates that require less deduction steps have less influence on the final valuation. This strategy has been motivated by the authors of NLRL to avoid local optima during the training phase. In summary, a_{t+1} is defined by applying all the

possible k set of rules that can jointly define a predicate p, weighting the results by W_p^*. Like δILP, NLRL sums the resulting valuations for each predicate p because the valuations are disjoint.

The architecture of NLRL can handle uncertain environments because each ground atom has a truth value in $[0, 1]$. Weights are iteratively updated to take the optimal action in the current state. A the end of the training phase, the rules with higher weights are presented to the user. NLRL is capable of using the learned rules to solve slightly different versions of the same task. The RL tasks that have been used to test the performance of SD-NLRL with respect to NLRL are presented in the following section.

5.3 RL tasks

RL methods are often validated using different type of problems. Unfortunately, a standard set of RL problems for neural-symbolic algorithms is not available. For example, the games for the Atari 2600 consoles are widely used to measure the performance of DRL algorithms. These environments do not provide for logical interface, and they cannot be directly used with neural-symbolic methods such as NLRL. In order to perform a fair comparison between NLRL and SD-NLRL, the set of tasks that are discussed in the paper that proposed NLRL were used. The tasks are five: CLIFFWALKING, WINDYCLIFFWALKING, ON, STACK, and UNSTACK.

In CLIFFWALKING, the environment is a 5 by 5 grid, and the agent is required to go from a start cell to a goal cell. When the agent goes onto a cliff cell, it receives a penalty of -1. Conversely, when the agent reaches the goal cell, it receives a reward of +1. Moreover, the agent is punished with a penalty of -0.02 every time it makes a move without reaching neither the goal cell nor the cliff cells. This small penalty is used to encourage the agent to reach the goal state taking the shortest path. Finally, the agent must complete the task within 50 steps, otherwise the game is terminated. WINDYCLIFFWALKING is a variant of CLIFFWALKING in which there is a 10% chance that the agent moves downward, no matter which is the action that it takes.

A state of the environment is described using a single predicate: `current(X,Y)`, which specifies the position of the agent in the grid. As discussed previously, the base problem makes use of a 5 by 5 grid. In order to correctly represent the state, the set of constants is $\{0, 1, 2, 3, 4\}$, and the background knowledge is composed of three predicates: `zero(X)`, `last(X)`, and `succ(X,Y)`. The predicates `zero(X)` and `last(X)` are used to refer to the constants representing the smallest number and the largest number, respectively. `succ(X,Y)` is used to define an ordering among constants (`succ(0,1)`,`succ(1,2)`,...).

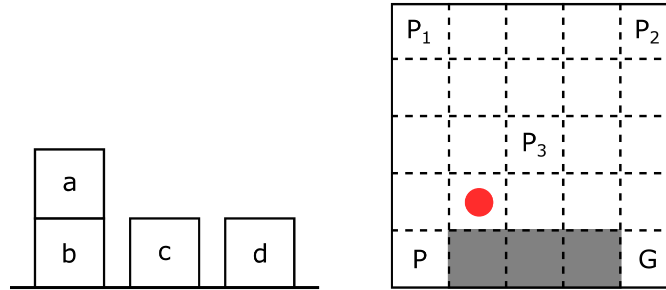

Figure 5.1: A block manipulation environment (on the left) and a cliff-walking environment (on the right).

Finally, the action atoms that are used by the agent to make a move on the grid are: `up()`, `down()`, `left()`, and `right()`.

The agent is trained on the base environment, and it is than tested using different versions of the environment to measure the generalization capability of the learning algorithm. Five variation of CLIFFWALKING and WINDYCLIFFWALKING are used: *top-left*, *top-right*, *centre*, *6x6*, and *7x7*. Top-left, top-right, and centre define the initial position of the agent as top left, top right, and the center of the grid, respectively. 6x6 and 7x7 define a larger grid of 6 by 6 and 7 by 7, respectively. Note that an increase of the grid size implies increasing the set of constants as well. For example, a state of CLIFFWALKING is represented in Figure 5.1. G represents the goal cell, and P, P_1, P_2, P_3 represent the starting cells of the agent. The red circle represents the current position of the agent, and the state can be described using the following sets of atoms:

```
[current(1,3)],
```

while the background knowledge can be represented as follows:

```
[zero(0), last(4), succ(0,1), succ(1,2), succ(2,3), succ(3,4)].
```

In STACK, UNSTACK, and ON the agent is required to manipulate blocks. In particular, STACK requires the agent to pile up all the blocks to obtain a single column. UNSTACK requires the agent to move all the blocks on the floor. ON requires the agent to move a specific block onto another specific block. Notably, only the block that is the top of a column can be moved. The agent gets a reward of $+1$ when it completes the task within 50 steps, otherwise the game is terminated. When the agent makes a move that does not end the game, it receives a small penalty of -0.02.

A state of the environment is represented using two predicates: `on(X,Y)` and `top(X)`. `on(X,Y)` specifies that a block X is on top of Y, where Y can be another block or the

floor. `top(X)` indicates that X is the top block of a column. Background knowledge is composed of a single predicate, `floor(X)`, that specifies which constant is used to represent the floor. For the ON task, the additional background predicate `goal(X,Y)` is used to express which block X must be on top of block Y. The set of constants is $\{a, b, c, d, floor\}$, and there is only one action predicate: `move(X,Y)`, where X must be a top block and Y must be another top block or the floor. When the agent takes an invalid action, the state of the environment is not changed. For example, a state representing a column of two blocks and two columns of one block is represented in Figure 5.1. The horizontal line represents the floor and the background knowledge can be described using the following sets of atoms:

```
[floor(floor)],
```

while the state can be represented as follows:

```
[top(a), on(a,b), on(b,floor), top(c), on(c,floor), top(d), on(d,floor)].
```

The base environment of UNSTACK starts with a single column of blocks. Five variants of the task are defined: *swap-2*, *divide-2*, *5-blocks*, *6-blocks*, and *7-blocks*. swap-2 and divide-2 swap the top two blocks and divide the blocks in two columns, respectively. The other three variants increase the number of blocks. The base STACK environment starts with all the blocks on the floor. Five variations of the task are defined: *swap-2*, *divide-2*, *5-blocks*, *6-blocks*, and *7-blocks*. swap-2 and divide-2 swap the two blocks on the right and divide the blocks in two columns, respectively. Similarly to the UNSTACK problem, the other three variants increase the number of blocks. The base environment of ON starts with a single column of blocks. Similar to UNSTACK and STACK, three variants are defined with an increasing number of blocks: *5-blocks*, *6-blocks*, and *7-blocks*. Two additional variants of the task are *swap-2* and *swap-middle*, which swap the top two blocks and the middle two blocks, respectively.

5.4 Inducing rules from states

One of the main issues of NLRL is that rules are generated using a top-down procedure and the user must specify a large amount of domain-specific details to successfully train the agent. In order to solve this problem, a new algorithm is proposed here. For the sake of clarity, it will be called SD-NLRL.

The idea behind SD-NLRL is to change the rule generation procedure f_{generate} to learn candidate rules directly from the states of the environment. In particular, the

algorithm assumes that the environment is able to provide the representations of all states. SD-NLRL makes use of the such representations to generate the candidate rules. This bottom-up approach is inspired by the classic bottom-up ILP algorithms such as Progol [67] (the algorithm combines top-down and bottom-up), which uses a technique called *bottom clause construction*. This technique starts defining the most specific rule that entails an example, and it then tries to generalize the rule to entail more examples. Bottom-up ILP methods start from many bottom clauses, and they try to iteratively obtain a small set of rules that represent the solution of the task. Here, SD-NLRL uses a slightly different approach because it generates a bottom clause from each state, but, it does not try to iteratively build the solution of the task. Instead, SD-NLRL tries to generate abstract rules that can represent a solution to the task, and it then uses the neural solver to select the subset of the rules that effectively solve the problem. Following the classification among top-down, bottom-up, and meta-level methods presented the previous chapter for ILP algorithms, SD-NLRL uses an hybrid approach that combines bottom-up and meta-level approaches.

SD-NLRL does not require the user to specify a program template because it directly generates candidate rules from the states of the environment. The only hyper-parameter that is required by SD-NLRL is the number of forward chaining steps T. Formally, SD-NLRL tries to predict the following probability:

$$P(\gamma|W,\Pi,L,B,S) = f_{\text{extract}}(f_{\text{infer}}(f_{\text{convert}}(B), f_{\text{generate}}(S,L), W, T), \gamma), \tag{5.21}$$

where f_{generate} takes a language L and the set of possible states S. SD-NLRL expects that each state is represented as a set of atoms. For each state $s \in S$ and each action atom $a \in A$, a bottom clause is built. In particular, the algorithm generates a bottom clause using the combination of the logical representation of the state and the associated background knowledge as the body of the clause, and of the action atom as the head of the clause. For example, assuming the following state from the CLIFFWALKING task: `[current(3,4)]`, the algorithm retrieves the atoms that are part of the background knowledge (they are included in L) and that share at least one term with the state atoms. For the above example the state representation becomes:

```
[current(3,4), succ(3,4), succ(2,3), last(4)],
```

because SD-NLRL finds all the atoms of B that contain terms 3 and 4, and it adds those atoms to the final state representation. Then, a ground rule is built for each action atom. The algorithm produces the following rules:

```
up() :- current(3,4), succ(3,4), succ(2,3), last(4),
```

```
down() :- current(3,4), succ(3,4), succ(2,3), last(4),
right() :- current(3,4), succ(3,4), succ(2,3), last(4),
left() :- current(3,4), succ(3,4), succ(2,3), last(4).
```

where `up()`, `down()`, `left()`, and `right()` are the action atoms that are defined as part of the task. Finally, SD-NLRL transforms the ground rules replacing the ground terms with variables. It introduces a new variable name every time it finds a new ground term, starting from the body of the rule and proceeding from left to right. Then, the algorithm does the same with the head of the rule. The ground rules are rewritten as:

```
up() :- current(X,Y), succ(X,Y), succ(Z,X), last(Y),
down() :- current(X,Y), succ(X,Y), succ(Z,X), last(Y),
right() :- current(X,Y), succ(X,Y), succ(Z,X), last(Y),
left() :- current(X,Y), succ(X,Y), succ(Z,X), last(Y),
```

where the variable names are taken from a list `[X,Y,Z,K,M]`. It would be possible to invert the valuation order in the procedure, starting from the head of the rule and then replacing the constants of the body. However, the resulting rules would be less readable because the head of the clause represents the consequence of the body. Therefore, it is more natural to define a substitution set for the body, and then use the substitution set for the head of the rule. For example, when SD-NLRL generates bottom clauses from the state in Figure 5.1, the algorithm starts from the following state:

```
[top(a), on(a,b), on(b,floor), top(c), on(c,floor), top(d), on(d,floor)].
```

There are 25 action atoms for each state: one for each pair of constants that are defined as part of the task. Assuming that the algorithm is trying to generate the clause for atom `move(a,c)`, which is defined as part of the task, it would build the following rule:

```
move(a,c) :- top(a), on(a,b), on(b,floor), top(c), on(c,floor), top(d),
             on(d,floor), floor(floor).
```

Then, SD-NLRL replaces the constants with variables, and it obtains the following rule:

```
move(X,K) :- top(X), on(X,Y), on(Y,Z), top(K), on(K,Z), top(M), on(M,Z),
             floor(Z)
```

Replacing the constants of the head before applying the same procedure to the body produces the following rule:

```
move(X,Y) :- top(X), on(X,Z), on(Z,K), top(Y), on(Y,K), top(M),
             on(M,K), floor(K)
```

Therefore, the head atom of the generated rules would be either `move(X,Y)` or `move(X,X)`, and the body of rules, which are obtained from the same states, would be completely different. For the sake of clarity, the algorithm that has been described here will be called `SD-NLRL unground`. The procedure is completely described by Algorithms 1 and 2. In particular, `unground_atom` takes as argument an atom a, a set of constants F, a set of variables E, and a substitution set μ. The algorithm starts defining the next possible variable, that is selected from E using function `get_next_var`, and the set of resulting terms *new_terms*. Algorithm 1 makes use of function `terms` to get all the terms contained in an atom. Moreover, it uses function `get_var` to check if the input term is a variable. If the set of constants F is non-empty, the algorithm transforms only the elements of F into variables. Otherwise, the algorithm transforms each term defined as a part of the atom. In particular, if F is empty, the algorithm considers each term, and if the current term is a variable, it is simply added to the resulting set of terms. Otherwise, if the term is not present in the substitution set μ, a new entry is created in μ and the term is added to *new_terms*. If F is non-empty, the algorithm simply adds a term to *new_terms*, when the term is a variable or when a term is not contained in F. Otherwise, if the term is contained in F, the algorithm adds the term to *new_terms*, and it updates the substitution set.

In order to reduce the number of generated rules, SD-NLRL enforces some of the constraints that are used also by NLRL:

1. Constants are not allowed in the generated rules;
2. All variables that appear in the head of a rule must appear also in its body;
3. Two rules that differ only in the order of the atoms in their bodies are not allowed together; and
4. An atom is not allowed at the same time in the head and in the body of a generated rule.

The above constraints are useful to reduce the number of the generated rules without reducing the flexibility of the algorithm.

An implicit consequence of using the new rule generation strategy is that SD-NLRL allows an action predicate to be defined by an unpredetermined number of rules. Moreover, the body of each candidate rule is composed of an arbitrary number of atoms. In order to support the changes of the rule generation process, the learning algorithm has been coherently modified. In particular, the set of generated rules for predicate p, cl_p, contains a single set of rules with $|cl_p| = d$. For each predicate p, SD-NLRL defines a

vector of weights $W_p \in \mathbb{R}^d$, which are normalized using a softmax function to obtain $W_p^* \in [0,1]^d$. The set of normalized weights is denoted as W.

The removal the constraints on rule generation improves the flexibility of the learning process, but it requires a larger amount of computational resources. In fact, SD-NLRL extends the definition of functions $F_c : [0,1]^n \to [0,1]^n$ to allow for an arbitrary number of body atoms. For each rule c, the algorithm defines a matrix $X_c \in N^{n \times w \times d}$, which is formally defined as:

$$x_k = \{(a_1, \dots, a_d) \mid \text{satisfies}_c(\gamma_{a_1}, \dots, \gamma_{a_d}) \wedge head_c(\gamma_{a_1}, \dots, \gamma_{a_d}) = \gamma_k\}$$
$$X_c[k,m] = \begin{cases} x_k[m] & \text{if } m < |x_k| \\ (0, \dots, 0) & \text{otherwise} \end{cases} \tag{5.22}$$

SD-NLRL prepares the matrices X_c before the training phase, and it takes d slices of X_c during the training process, namely, $X_1, \dots, X_d \in \mathbb{N}^{n \times w}$. X_i represents the i-th element of the body of clause c. Then, the algorithm uses the gather$_2$ function defined in 5.14 to obtain matrices $Y_1, \dots, Y_d \in [0,1]^{n \times w}$, which represent the current fuzzy truth values of a valuation that are referenced by $X_1, \dots, X_d$. SD-NLRL makes use of $Y_1, \dots, Y_d$ to define the matrix $Z_c \in [0,1]^{n \times w}$ as:

$$Z_c = Y_1 \odot \dots \odot Y_d \tag{5.23}$$

Finally, the algorithm defines F_c in the same way as NLRL:

$$F_c(a) = a' \quad \text{where} \quad a'[k] = max(Z[k, _]) \tag{5.24}$$

In SD-NLRL, a forward chaining step is defined in a similar way to NLRL:

$$a_{t+1} = a_0 + \left(\sum_{p \in \text{Pred}_i} \sum_j^{\oplus} W_p^*[j] F_p^j(a_t) \right) \tag{5.25}$$

where $W_p^*[j]$ is the weight associated to the j-th rule defined for predicate p, and F_p^j performs a deduction step for the j-th rule defined for predicate p.

The experimental results that were obtained from the study of the first prototype of SD-NLRL show that the generated rules are overly specific. In particular, the algorithm is able to learn a model that solves the base task of CLIFFWALKING, but it fails to generalize to the variants of the same task. Moreover, it often fails to to learn a model using even the standard environment. The algorithm learns a completely wrong strategy, and it remains stuck in a local optima until the end of the training. This problem has been solved for the second prototype, and it is discussed in the next section. Another

Algorithm 1 The function to transform a ground atom into an atom containing variables

```
1: function UNGROUD_ATOM(a, F, E, μ)
2:     current_var ← get_next_var(E)
3:     new_terms ← {}
4:     if |F| = 0 then
5:         for each term ∈ terms(a) do
6:             if is_var(term) then
7:                 new_terms ← new_terms ∪ term
8:             else
9:                 if term ∉ μ then
10:                    μ ← μ ∪ {current_var/term}
11:                    current_var ← get_next_var(E)
12:                end if
13:                new_terms ← new_terms ∪ μ[term]
14:            end if
15:        end for
16:    else
17:        for each term ∈ terms(a) do
18:            if is_var(term) or term ∉ F then
19:                new_terms ← new_terms ∪ term
20:            else
21:                if term ∈ F then
22:                    new_terms ← new_terms ∪ term
23:                    μ ← μ ∪ {current_var/term}
24:                    current_var ← get_next_var(E)
25:                end if
26:            end if
27:        end for
28:    end if
29:    return a new atom with terms new_terms
30: end function
```

Algorithm 2 The function to transform a ground rule into a rule containing variables

1: **function** UNGROUD(h, b, F)
2: $\mu \leftarrow \{\}$
3: $\text{final}_b \leftarrow \{\}$
4: $\text{ext}_v \leftarrow \text{vars}(h) \cup \text{vars}(b)$
5: **for each** atom $\in b$ **do**
6: res $\leftarrow$ unground_atom$(\text{atom}, F, \text{ext}_v, \mu)$
7: $\text{final}_b \leftarrow \text{final}_b \cup \text{res}$
8: **end for**
9: $\text{final}_h \leftarrow$ unground_atom$(h, F, \text{ext}_v, \mu)$
10: **return** a new rule with head final_h and body final_b
11: **end function**

problem is that the algorithm requires a large amount of computational resources. In fact, SD-NLRL must perform each deduction step using large matrices. Moreover, it generates a large amount of candidate clauses that further increases the complexity of the learning process. This problem has prevented the first prototype of the algorithm from learning the block manipulation tasks. In particular, SD-NLRL generates 470 candidate rules for the STACK and UNSTACK environments, and it generates 1488 and 84 rules for the ON and CLIFFWALKING environments, respectively. The number of generated rules and the size of their bodies for the ON, STACK, and UNSTACK environments are too high, and the algorithm was not able to complete the training process. Moreover, the candidate rules should be less state-specific, and the number of body atoms should be reduced as much as possible. In order to improve the performance of SD-NLRL, as well as its demand of computational resources, the rule generation procedure was changed to produce a small number of rules. Having a smaller set of general candidate rules allows the SD-NLRL neural solver to efficiently find a solution to the considered tasks.

5.5 Inducing general rules from states

Finding a good set of candidate rules is important to increase the generalization capabilities of the proposed algorithm and to reduce the required computational resources. In order to improve the quality of the generated rules, SD-NLRL tries to reduce both the number of atoms in the body of the learned rules and the number of candidate rules.

The logical representation of the states describes the interesting and characterizing features of the states. Many RL algorithms extract high-level features from the states,

Algorithm 3 The function that performs rule generation

```
1: function GEN_RULES(A, S, B)
2:     rules ← {}
3:     for each s ∈ S do
4:         G ← get_groups(s)
5:         for each g ∈ G do
6:             for each a ∈ A do
7:                 shared ← get_shared(a, g)
8:                 for each b ∈ B do
9:                     c ← constants(b)
10:                    if c ∩ shared ≠ ∅ then
11:                        g ← g ∪ b
12:                    end if
13:                end for
14:                free ← get_free(a, g)
15:                if |free| = 0 then
16:                    r ← unground(a, g, {})
17:                    rules ← rules ∪ r
18:                else
19:                    for each f ∈ free do
20:                        fixed ← free \ {f}
21:                        partial_rule ← unground(a, g, fixed)
22:                        for each b ∈ B do
23:                            if f ∈ constants(b) then
24:                                body(partial_rule) ← body(partial_rule) ∪ b
25:                            end if
26:                        end for
27:                        r ← unground(a, body(partial_rule), {})
28:                        rules ← rules ∪ r
29:                    end for
30:                end if
31:            end for
32:        end for
33:    end for
34:    return rules
35: end function
```

and DRL algorithms are a clear example of this strategy. In particular, when the state is represented by an image, these algorithms employ a convolutional neural network to extract the most relevant features of the image, which are then used to pick the best possible action. The second prototype of SD-NLRL tries to follow the same strategy, and it splits the logical representation of each state into different groups of atoms, which represent the logical features of the states. The algorithm is completely described in Algorithm 3. The procedure `gen_rules` takes three arguments: the set of action atoms A, the set of states S, and the background atoms B. The procedure starts defining the set of generated rules, namely *rules*, and each state s is subdivided into one or more groups of atoms. Note that two different groups must have completely different sets of constants. The function `get_groups` implements the subdivision of the state into groups, and it returns a list of groups G. Then, at least one new rule is defined for each group of atoms and for each action atom. Therefore, a single state induces a set of rules that can simultaneously be true. Each rule represents a logical feature of the state, and it contributes to the truth value of the target predicate. The constants that are shared between the action atom and the body of the rule are the most significant ones because they relate the features of the state to the action to be taken. Therefore, SD-NLRL includes in each group every background atom that shares a constant with the group. The procedure `gen_rules` obtains the set of shared constants, namely *shared*, using the function `get_shared`. Then, it obtains the set of constants of each background atom b using the function `constants`, and it adds the background atom to the group if b contains at least one of the shared constants. Finally, the algorithm builds the set of free constants, which are the constants that are not shared among the action atom and the body of the rule after the inclusion of the correlated background atoms. In the pseudo-code, this procedure is represented by the function `get_free`. The algorithm then checks if the set of free constants, namely `free`, is empty. If *free* is empty, there is only one rule that can be generated. Therefore, SD-NLRL directly applies the `unground` function, and the resulting rule is inserted in the set of generated rules. On the contrary, if the set of free constants is non-empty, the algorithm generates a different rule for each free constant f. In particular, the algorithm applies the `unground` function to transform the free constants, except f, into variables. Then, it includes background atoms that contain f in the body of the generated rule. The function `body` is used to obtain the body atoms of an input rule. Finally, the algorithm applies another time the function `unground` to transform the remaining constants into variables. In summary, SD-NLRL produces one rule for each group of correlated atoms, for each action atom, and for each free constant, adding the needed background knowledge whenever necessary. Therefore, each action

atom is defined by one or more rules that contribute to its truth value depending on the associated weight and on the specific features that the state presents. For example, when generating the rules from the following state of CLIFFWALKING task:

```
[current(2,3)],
```

SD-NLRL generates a single group containing the previous atom only. Then, for each action atom, it tries to add background atoms that contain constants that are shared between the action atom and the atom in the group. Unfortunately, the action atoms have an arity of 0, and the algorithm is not able to add any background atom to the group. Then, for each free constants, i.e. 2 and 3, SD-NLRL applied the `unground` function to the non-free constants, and it includes the background atoms that contain the free constant. Therefore, the algorithm generates the following rules:

```
ACTION() :- current(2,X),succ(1,2),succ(2,3),
ACTION() :- current(X,3),succ(2,3),succ(3,4),
```

where `ACTION()` can be replaced with `up()`, `down()`, `left()`, or `right()`. Finally, SD-NLRL applies the `unground` function, and it generates the following rules:

```
ACTION() :- current(Y,X),succ(Z,Y),succ(Y,M),
ACTION() :- current(X,Y),succ(Z,Y),succ(Y,M).
```

Here, variables `X` and `Y` can be assigned to the same value. Therefore, the generated rules are true in the state `current(2,3)`, but they are true in other states such as `current(1,2)`.

The current implementation of SD-NLRL is based on the existing implementation of NLRL, which is available on GitHub[2]. The implementation of NLRL has not been updated in the last three years, and it is based on Python 2.7 and Tensorflow 1.11. Unfortunately, these technologies are now considered as deprecated, but the implementation of SD-NLRL has been obtained by modifying the implementation of NLRL to prevent any implementation detail from altering the comparison between the two methods.

SD-NLRL shares many hyper-parameters with NLRL. In particular, Table 5.1 shows the hyper-parameters that are shared between NLRL and SD-NLRL. Moreover, the implementation of NLRL defines the number of forward chaining steps T as 4, and the learning rate as 0.001. The number of training steps are 50,000 for CLIFFWALKING and WINDYCLIFFWALKING, and they are 30,000 for the block manipulation tasks. Many of the used hyper-parameters that are defined in the implementation are equal

[2] `https://github.com/ZhengyaoJiang/NLRL`

Table 5.1: The hyper-parameters that are shared between NLRL and SD-NLRL.

Hyper-parameter	Value
Number of steps per batch	10
γ	1
λ	0.95
Optimizer	RMSProp
RMSProp α	0.9
RMSProp ϵ	0.0000000001
RMSProp momentum	0
Size of the hidden layer of the critic	20 units

to the ones that have been reported by the paper that introduces NLRL. The original paper defines four auxiliary predicates that share common characteristics in all the tasks. Moreover, the paper defines the action predicates for each task. The actual implementation of NLRL makes use of the same predicate definitions reported by the original paper. In particular, the auxiliary predicates are defined using the following program template:

```
⟨[inv1,inv2,inv3,inv4], [2,2,1,1], [[(1,True)],[(1,True)],[(1,True)],
[(2,False)]], 4⟩,
```

where the first and the second parameters of the template are vectors containing the names and the arities of the auxiliary predicates, respectively. The last parameter is the number of deduction steps T, and the third parameter is a vector containing the rule templates of the corresponding predicates. NLRL defines the templates of the action predicates for CLIFFWALKING and WINDYCLIFFWALKING as $\langle 0, [(3, True)]\rangle$, where the first argument is the arity of the predicate, and the second argument is a vector containing the rule templates. The templates of the action predicate for STACK and UNSTACK are defined as $\langle 2, [(1, True)]\rangle$, while the template for ON is defined as $\langle 2, [(1, True), (0, True)]\rangle$. The discussed program templates define the rules generated for each action predicate. For example, the following rules are generated using the program template of action predicate `up()` for CLIFFWALKING:

```
up() :- succ(X,Y),succ(Y,Z),
up() :- succ(X,Y),inv1(Z,Y).
```

It is worth noting that the above rules presents 3 free variables, and each rule can contain both intensional (`inv1`) or extensional (`succ`) predicates.

Unlike NLRL, SD-NLRL does not require to specify a template for each predicate. SD-NLRL requires to specify only the arity of the action predicates. Moreover, in the second prototype of SD-NLRL, T is defined as 1 for all the tasks because the algorithm generates rules that do not use auxiliary predicates. Therefore, a single forward chaining step is sufficient to correctly compute the fuzzy truth value of the action predicates. The reduction of the number of deduction steps can considerably improve the learning speed as well as the time required to pick an action during the evaluation phase.

The first prototype of SD-NLRL uses a softmax function to normalize the weights. However, different normalization techniques can be applied to the weights. Moreover, p_A can be changed to modify the strategy that is used to select the action. Initially, the algorithm used a softmax function to normalize the weights, and it used the same action selection strategy that has been previously discussed. The learning rate was initially set to 0.001, which is the same value that was proposed for NLRL. Unfortunately, the updated rule generation strategy was not able to reduce the number of rules for the block manipulation tasks. In fact, the generated rules for these tasks present multiple atoms that are connected following a recursive pattern, which cannot be captured by the proposed strategy. For example, the state:

```
[top(a), on(a,b), on(b,c), on(c,d), on(d,floor)],
```

which represents a single column of four blocks, cannot be subdivided into different groups of atoms because each atom is directly, or indirectly, connected to the other atoms. Moreover, the only background atom that can be added to the list of atoms is `floor(floor)`. Therefore, the background atom is always inserted in the list, and the final set of generated rules is the same as the one generated by the first prototype of SD-NLRL. As a consequence, the second prototype of SD-NLRL was not able to complete the training process for the block manipulation tasks.

In order to test the performance of the second prototype, 10 runs have been performed for both the CLIFFWALKING and WINDYCLIFFWALKING tasks. Unfortunately, the algorithm struggled to learn an effective strategy. In fact, the action selection strategy p_A was designed to increase, as much as possible, the difference between the weights of the best rules and the weights of the other rules. NLRL generates a large number of candidate rules. Therefore, it takes a considerable amount of time to significantly change the rule weights, and the algorithm is able to randomly get a significant amount of positive rewards during this time. On the contrary, SD-NLRL generates a small number

of rules for the two cliff-walking tasks. In particular, the algorithm generates 9 rules for each action predicate. The generated rules are:

```
ACTION() :- current(X,X),succ(X,Y),zero(X)
ACTION() :- current(Y,X),succ(Z,Y),succ(Y,M)
ACTION() :- current(X,Y),succ(Y,Z),zero(Y)
ACTION() :- current(X,X),last(X),succ(Y,X)
ACTION() :- current(X,Y),succ(Z,Y),succ(Y,M)
ACTION() :- current(Y,X),last(Y),succ(Z,Y)
ACTION() :- current(X,Y),last(Y),succ(Z,Y)
ACTION() :- current(X,X),succ(Y,X),succ(X,Z)
ACTION() :- current(Y,X),succ(Y,Z),zero(Y)
```

where `ACTION()` is a placeholder for `up()`, `down()`, `left()`, and `right()`. The algorithm generates only 36 candidate rules. Therefore, it is much more likely for the agent to remain trapped into a local optimum as the algorithm tends to immediately reward a strategy that is not the correct solution to the problem. As a consequence, the algorithm often learns a model that performs worse than the random player. If the algorithm receives a sufficient amount of positive rewards during the initial phase of the training, it is able to learn a strategy that can be used to effectively complete the task. Otherwise, the agent learns a completely wrong strategy, and, as the training progresses, it fails to change the rule weights. For example, the agent often learned to go to a cliff cell. Therefore, it was not able to randomly reach the goal cell, and this behaviour makes the agent reinforce the current strategy. The learning rate is continually decreased during training. At the same time, the neural critic continually improves the accuracy of the estimated state-value function. The updates to the weights are thus increasingly smaller as the training progress, and, at some point, the agent completely stops learning.

Another problem with the initial configuration of the prototype was the softmax function used to normalize the weights. In fact, the softmax function makes the sum of the fuzzy truth values of the weights equal to 1. Therefore, as the learning progresses, only 1 rule is selected for each action predicate. In order to solve this problem, a standard normalization is applied to the weights. In particular, each weight $W_p[j]$ is normalized using the following rule:

$$W_p^*[j] = \frac{W_p^{max} - W_p[j]}{W_p^{max} - W_p^{min}} \tag{5.26}$$

where $W_p^*[j]$ denotes the normalized weight, W_p^{max} denotes the maximum weight defined for predicate p, and W_p^{min} denotes the minimum weight defined for predicate p. Using

Table 5.2: A performance comparison between NLRL and SD-NLRL on the CLIFFWALKING and WINDYCLIFFWALKING tasks. The optimal returns (Optimal column) are taken from the paper that introduces NLRL [9].

Env.	Task	NLRL	NLRL$_p$	SD-NLRL	Optimal
CLIFF	train	0.807 ± 0.048	0.862 ± 0.026	0.673 ± 0.313	0.880
	top-left	0.763 ± 0.021	0.749 ± 0.057	0.647 ± 0.206	0.840
	top-right	0.829 ± 0.020	0.809 ± 0.064	0.788 ± 0.031	0.920
	centre	0.850 ± 0.018	0.859 ± 0.050	0.752 ± 0.257	0.920
	6x6	0.778 ± 0.062	0.841 ± 0.024	0.576 ± 0.499	0.860
	7x7	0.765 ± 0.029	0.824 ± 0.024	0.529 ± 0.553	0.840
WINDY	train	0.135 ± 0.481	0.663 ± 0.377	-0.806 ± 0.282	0.769 ± 0.162
	top-left	0.250 ± 0.731	0.726 ± 0.075	-0.501 ± 0.459	0.837 ± 0.068
	top-right	0.346 ± 0.730	0.834 ± 0.061	-0.319 ± 0.555	0.920 ± 0.000
	centre	0.498 ± 0.504	0.672 ± 0.579	-0.375 ± 0.526	0.868 ± 0.303
	6x6	0.192 ± 0.633	0.345 ± 0.736	-0.863 ± 0.239	0.748 ± 0.135
	7x7	0.173 ± 0.660	0.506 ± 0.528	-0.875 ± 0.247	0.716 ± 0.181

this normalization technique, a predicate can be defined by multiple rules because W_p^{max} and W_p^{min} represent the only constraints on the weights.

This normalization technique allows a predicate to be defined by multiple rules. However, the training process becomes unstable because a large update to W_p^{max} or W_p^{min} results in a large update to every other weight. As a consequence, maintaining a learning rate of 0.001, the agent often learns a wrong strategy, in the same way that have been discussed before. However, the algorithm keeps changing the weights performing very small updates. The final result is the same but, in this case, the values of the weights constantly oscillate. Basically, this problem is caused by the learning rate being too large. The best performance is obtained combing this normalization technique with an appropriate learning rate. In particular, the value of the learning rate depends on the complexity of the task. A learning rate value of 0.0005 has been selected to obtain the best performance on the CLIFFWALKING task. The WINDYCLIFFWALKING task is more difficult, and the agent obtains a positive reward less frequently in the initial phase of the training. In fact, the most appropriate value of the learning rate that has been tested is 0.0001, which is significantly smaller than the value that has been used for

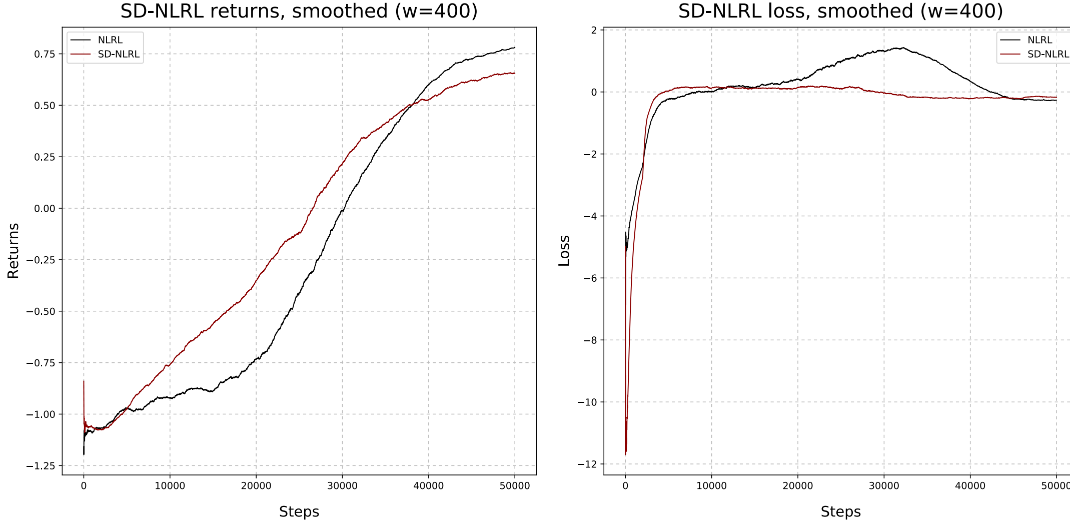

Figure 5.2: Average returns (on the left) and losses (on the right) for CLIFFWALKING.

CLIFFWALKING. It is worth noting that some tests have been performed by changing the action selection strategy p_A. In particular, the previously discussed strategy has been replaced by a softmax function. However, the resulting probabilities were biased because the softmax was applied to values in $[0, 1]$. Therefore, the range of possible values of the output probabilities was smaller than $[0, 1]$, and the agent was not able to learn and behave correctly in the cliff-walking tasks.

Table 5.2 reports a comparison between the performance of NLRL and the performance of the second prototype of SD-NLRL for the two cliff-walking tasks. The table contains also the column NLRL_p that shows the performance of NLRL as reported by the the original paper [9], for completeness. It is worth noting that the performance reported by the original paper is much better than the performance that the used implementation of NLRL shows. The original paper does not report the number of trainings that have been performed to obtain the corresponding results. On the contrary, the information that is presented in the table has been produced making 3 trainings for each task for NLRL and 5 trainings for each task for SD-NLRL. In fact, it was impossible to make more trials for NLRL because the algorithm requires a considerable amount of time to complete a training process (NLRL requires approximately 10 days for each training while SD-NLRL requires less than 2 hours). Therefore, the difference between the results can be justified by the small number of trainings that have been made using the actual implementation of NLRL. In order to measure the performance of both methods, each trained model has been evaluated on 100 complete episodes. Then, the

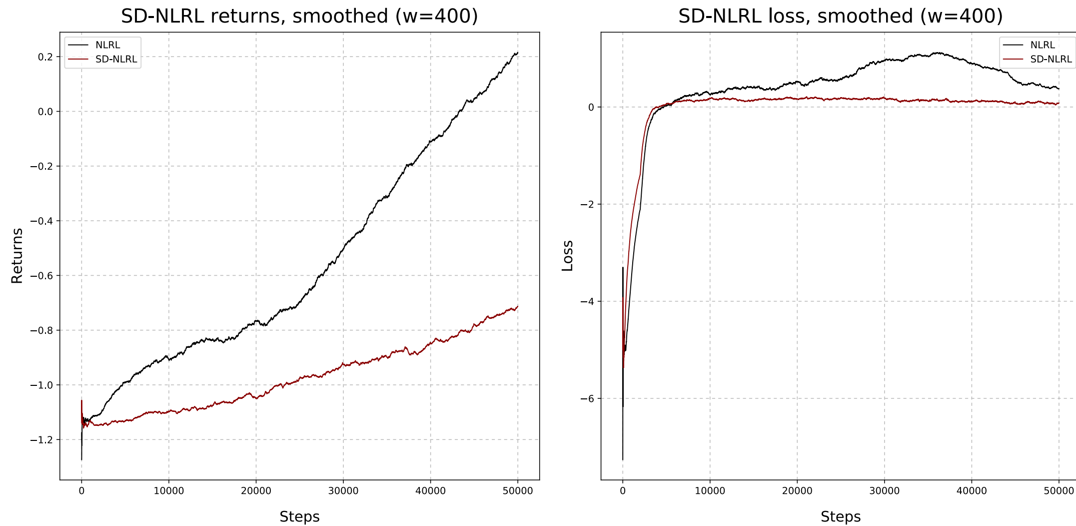

Figure 5.3: Average returns (on the left) and average losses (on the right) for the WINDY-CLIFFWALKING task.

average return has been computed using the returns that the agent received during the evaluation phase. Finally, the values that are shown in the table have been computed by averaging the average returns on the learned models. Table 5.2 shows that NLRL slightly outperforms SD-NLRL on the CLIFFWALKING task. It is worth noting that the performance of SD-NLRL on CLIFFWALKING suffers from a great variance. In fact, in 1 of the 5 trainings, SD-NLRL learned a strategy that is able to occasionally get a positive reward but that is far from the optimal strategy. In particular, the agent stopped learning, and the algorithm reached a local optimum. This behaviour suggests that the training process can be particularly sensible to the number of generated rules. The algorithm should be able to continually explore new action-state pairs avoiding local optima even when the number of rules is very small, and solving this problem represents an important goal for the future. The performance of SD-NLRL on the WINDYCLIFFWALKING task is far from the performance of NLRL, and the proposed algorithm was not able to learn an effective strategy to complete the task. It is worth noting that the algorithm sometimes get a positive reward, and it was able to learn over time. Unfortunately, the learning speed is very slow, and the algorithm often remains trapped into a local optimum. Figure 5.2 and Figure 5.3 show the average losses and average returns for the CLIFFWALKING task and the WINDYCLIFFWALKING task, respectively. In particular, the figures have been computed using the training data (only the training environment has been considered). In order to generate the figures, the losses

and the returns have been averaged on the learned models. Then, the resulting values have been averaged every 5 values to improve the readability of the graphs. Finally, each curve has been smoothed by averaging the points using a sliding window of $w = 400$. Despite SD-NLRL reaching lower average returns than NLRL, both the loss curves and the return curves are smoother in the case of SD-NLRL. Therefore, the algorithm is able to constantly learn, but it reaches a lower overall performance. In NLRL, the rules are generated using carefully hand-crafted templates that produce sufficiently general rules. SD-NLRL generates the rules from the states of the environment. Therefore, the generated rules can be overly specific, and the set of generated rules could not include a good strategy to complete the task.

SD-NLRL shows performance that are worse than the performance of the original algorithm, but it was able to perform coherently on tasks that are different from the one that has been used for training. Moreover, SD-NLRL does not require the user to specify the form of the final solution, which is very important. In fact, it is very difficult, or even impossible, to provide such information in many real-world tasks. However, the set of all possible states is not often available, and the removal of this limitation represents an important goal for future research.

The number of generated rules plays an important role from the perspective of the required computational resources. In particular, SD-NLRL generates only 36 rules, while NLRL generates 2813 rules. Therefore, NLRL requires a considerable amount of time to complete a training process on the cliff-walking tasks, compared to SD-NLRL. As previously mentioned, in the case of CLIFFWALKING, for example, NLRL requires approximatively 10 days to complete a training. SD-NLRL requires only less than two hours to perform the same training process and using the same machine.

5.6 Learning to avoid an action

In both the CLIFFWALKING and the WINDYCLIFFWALKING tasks, the agent can move in the four directions. However, the goal cell is always defined as the bottom-right cell of the grid. Therefore, considering the specific definitions of the tasks discussed in Section 5.3, the agent must always move to the right, before going down, to correctly complete the task. The set of learned rules that defines the predicate `left()` must be impossible to apply. The paper that introduces NLRL reports a similar situation, and the algorithm learns to avoid moving left. However, the paper shows that NLRL gives a high weight to a rule that cannot be used. On the contrary, in the experiments that have been made with the NLRL implementation, the agent learns to avoid moving left, but the

Table 5.3: The rules generated by the third prototype of SD-NLRL (on the right), with the associated weights (on the left).

Weight	Rule
0.14	`left() :- current(Y,X), succ(Y,Z), zero(Y)`
0.18	`left() :- current(X,Y), last(Y), succ(Z,Y)`
0.19	`left() :- current(X,X), succ(Y,X), succ(X,Z)`
1.0	`left() :- current(Y,X), succ(Z,Y), succ(Y,M)`

algorithm gives a very small weight to the rules. In fact, the weights of the learned rules for predicate `left()` are very small, and only a few weights slightly exceed 0. However, NLRL does not explicitly contemplate the possibility that one or more actions can be forbidden by the learned strategy.

The third prototype of SD-NLRL expects that each action can be forbidden. In particular, the algorithm inserts a special rule, which is called here *unreachable rule* for the sake of clarity, into the set of generated rules, for each action predicate. The unreachable rule has the following structure:

```
ACTION() :- unreachable_state(),
```

where `ACTION()` is a placeholder `up()`, `down()`, `left()`, and `right()`. The predicate `unreachable_state()` is a special predicate that must be different from all the predicates that are defined in the task. Inserting an unreachable rule can be useful from the perspective of the interpretability of the learned rules. In fact, the strategy that NLRL was able to learn is not easily readable. On the contrary, the unreachable rule represents a compact and readable solution to the problem.

In order to measure the quality of the generated rules, 5 trainings have been made for both the CLIFFWALKING and the WINDYCLIFFWALKING tasks. The evaluation procedure that has been used is the same discussed in the previous section. The experimental results shows that the third prototype of SD-NLRL completely ignores the unreachable rule in the CLIFFWALKING task, and the algorithm gives a weight of 0 to the unreachable rule in every test. Moreover, the overall performance of the prototype is even worse than the performance of the second prototype. The learned rules (with weight greater than 0.05) of the best trained model for predicate `left()` are shown in Table 5.3. The learned rules do not represent a good strategy because the agent goes left in many situations, though the correct strategy is to go right from almost all the

Table 5.4: A performance comparison between the second prototype of SD-NLRL (SD-NLRL$_2$) and the third prototype of SD-NLRL (SD-NLRL$_3$) on cliff-walking tasks. The returns for SD-NLRL$_3$ are restricted to the best trained model.

Environment	Task	SD-NLRL$_2$	SD-NLRL$_3$
CLIFF	train	0.673 ± 0.313	0.421
	top-left	0.647 ± 0.206	0.393
	top-right	0.788 ± 0.031	0.737
	centre	0.752 ± 0.257	0.615
	6x6	0.576 ± 0.499	0.211
	7x7	0.529 ± 0.553	-0.166
WINDY	train	-0.806 ± 0.282	-0.397
	top-left	-0.501 ± 0.459	-0.135
	top-right	-0.319 ± 0.555	0.280
	centre	-0.375 ± 0.526	0.262
	6x6	-0.863 ± 0.239	-0.521
	7x7	-0.875 ± 0.247	-0.630

positions. The performance of the algorithm for the WINDYCLIFFWALKING tasks is comparable to the one of the second prototype. The agent does not effectively learn a good strategy, and it occasionally gives a value greater than 0 to the unreachable rule. Table 5.4 shows a comparison between the performance of the second prototype of SD-NLRL and the performance of the best model obtained from the experiments on the third prototype of the algorithm. The comparison shows that, in the CLIFFWALKING task, the performance of best model that the third prototype of SD-NLRL learned is worse than the one obtained by the second prototype of the algorithm. In the case of WINDYCLIFFWALKING, the best model that is learned by the third prototype of SD-NLRL shows a performance that is similar to the one obtained by the second prototype of the algorithm. Learning to explicitly avoid certain actions is an interesting advance from the perspective of the interpretability of the learned rules, and further experiments are needed to understand how to achieve this goal.

5.7 Dealing with recursive patterns within states

Dealing with states that present a recursive structure is not a trivial task. In fact, bottom-up ILP methods typically struggle inducing recursive clauses from examples [53]. A typical example of a simple recursive pattern is the following:

```
p(X,Y),p(Y,Z),p(Z,K).
```

The variables in the above sequence connect the atoms following a chain pattern. This simple pattern can be captured by the following rules:

```
p2(X,Y) :- p(X,Z),p(Z,Y),
p2(X,Y) :- p(X,Z),p2(Z,Y).
```

In fact, starting with a list of atoms containing a chain pattern, it is possible to opportunely replace the occurrences of `p` with occurrences of `p2`. The obtained list of atoms is not equivalent to the original list, but it represents an arbitrarily long chain of occurrences of `p`.

The fourth prototype of SD-NLRL tries to capture the chain patterns included in states to reduce the number of generated rules as well as the number of atoms in the body of the generated rules. In particular, the prototype defines a special predicate defined as follows:

```
rec(X,Y) :- on(X,Z),on(Z,Y),
rec(X,Y) :- on(X,Z),rec(Z,Y).
```

The algorithm employs a modified version of the rule generation function shown in Algorithm 3. The modified rule generation function is described in Algorithm 4. The algorithm uses function `replace_chain` to find the longest sequence of atoms that follows a chain pattern, and to replace the sequence with `rec(A,B)`, where `A` and `B` are the constants that limit the sequence on the left and on the right, respectively. For example, the state:

```
[top(a), on(a,b), on(b,c), on(c,floor), top(d), on(d,floor)]
```

is transformed into the following state:

```
[top(a), rec(a,floor), top(d), on(d,floor)].
```

Then, the function `replace_chain` applies a sorting algorithm to the state. The algorithm decides the ordering of the atoms looking at the names of the predicates. When

Algorithm 4 The function that performs rule generation considering also recursive patterns

```
1: function GEN_RULES(A, S, B)
2:     rules ← {}
3:     for each s ∈ S do
4:         G ← get_groups(s)
5:         for each g ∈ G do
6:             for each a ∈ A do
7:                 shared ← get_shared(a, g)
8:                 for each b ∈ B do
9:                     c ← constants(b)
10:                    if c ∩ shared ≠ ∅ then
11:                        g ← g ∪ b
12:                    end if
13:                end for
14:                free ← get_free(a, g)
15:                if |free| = 0 then
16:                    g ← replace_chain(g)
17:                    r ← unground(a, g, {})
18:                    rules ← rules ∪ r
19:                else
20:                    for each f ∈ free do
21:                        fixed ← free \ {f}
22:                        partial_rule ← unground(a, g, fixed)
23:                        for each b ∈ B do
24:                            if f ∈ constants(b) then
25:                                body(partial_rule) ← body(partial_rule) ∪ b
26:                            end if
27:                        end for
28:                        body(partial_rule) ← replace_chain(body(partial_rule))
29:                        r ← unground(a, body(partial_rule), {})
30:                        rules ← rules ∪ r
31:                    end for
32:                end if
33:            end for
34:        end for
35:    end for
36:    return rules
37: end function
```

the names of the predicates are equal, the algorithm looks at the terms proceeding from left to right. The sorting algorithm is applied to further reduce the number of generated rules when `unground` is applied. For example, assuming that `unground` is applied to the following two states:

```
[top(a), rec(a,floor), top(d), on(d,floor)],
[top(a), top(c), rec(a,floor), on(c,floor)],
```

the bodies of the generated rules would be:

```
[top(X), rec(X,Y), top(Z), on(Z,Y)],
[top(X), top(Y), rec(X,Z), on(Y,Z)].
```

The generated rules would appear as different rules, but they represent the same situation. The sorting algorithm transforms the list of atoms, which would be defined as:

```
[on(X,Y), rec(Z,Y), top(Z), top(X)],
[on(X,Y), rec(Z,Y), top(Z), top(X)].
```

In this case, the generated rules are equal, and the algorithm discard the duplicate rules.

Table 5.5 shows the experimental results relative to the fourth prototype of SD-NLRL. In the training environments, SD-NLRL clearly outperforms NLRL in the block manipulation tasks when considering the experimental results relative to NLRL, and it reaches a performance that is slightly worse than the one reported by the paper that introduces NLRL. However, SD-NLRL generalizes only to some block manipulation tasks. In particular, when the number of constants increases the algorithm struggles to learn a good strategy. In the ON task, SD-NLRL fails to complete the evaluation process in two environments because the learned model exceeds the size limits of Tensorflow. The reasons behind this behaviours are unclear, and further experiments are required to completely understand this phenomenon. When the number of constants is not increased, the agent reaches a near-optimal performance in both STACK and UNSTACK, but it reaches a lower performance on ON. This is justified because ON is more difficult than STACK and UNSTACK. Figure 5.5, Figure 5.4, and Figure 5.6 show the average losses and returns for the block manipulation tasks. In particular, the figures have been computed using the data that is obtained during the training process (only the training environment has been considered). Both the numerical results in Table 5.5 and in the figures have been computed using the same methodology that is discussed in Section 5.5 for the cliff-walking tasks. The figures show that SD-NLRL learns faster than NLRL, and it reaches a better overall performance. At the same time, the loss rapidly reaches 0 as the

Table 5.5: A performance comparison between NLRL and SD-NLRL on the STACK, UNSTACK, and ON tasks. The optimal returns (Optimal column) are taken from the paper that introduces NLRL [9].

Env.	Task	NLRL	NLRL$_p$	SD-NLRL	Optimal
UNSTACK	train	0.869 ± 0.007	0.937 ± 0.008	0.934 ± 0.002	0.940
	swap-2	0.872 ± 0.003	0.936 ± 0.009	0.934 ± 0.001	0.940
	divide-2	0.919 ± 0.006	0.958 ± 0.006	0.955 ± 0.001	0.960
	5-blocks	0.786 ± 0.008	0.915 ± 0.010	0.076 ± 0.080	0.920
	6-blocks	0.660 ± 0.003	0.891 ± 0.014	-0.566 ± 0.074	0.900
	7-blocks	0.559 ± 0.021	0.868 ± 0.016	-0.891 ± 0.033	0.880
STACK	train	0.768 ± 0.015	0.910 ± 0.033	0.895 ± 0.047	0.940
	swap-2	0.769 ± 0.053	0.913 ± 0.029	0.896 ± 0.052	0.940
	divide-2	0.820 ± 0.028	0.897 ± 0.064	0.885 ± 0.047	0.940
	5-blocks	0.541 ± 0.040	0.891 ± 0.032	0.838 ± 0.064	0.920
	6-blocks	0.234 ± 0.069	0.856 ± 0.169	-0.316 ± 0.060	0.900
	7-blocks	-0.157 ± 0.135	0.828 ± 0.179	-0.867 ± 0.075	0.880
ON	train	0.626 ± 0.073	0.915 ± 0.010	0.801 ± 0.027	0.920
	swap-2	0.615 ± 0.091	0.912 ± 0.013	0.809 ± 0.027	0.920
	swap-middle	0.636 ± 0.083	0.914 ± 0.011	0.739 ± 0.015	0.920
	5-blocks	0.389 ± 0.134	0.890 ± 0.016	-0.096 ± 0.063	0.900
	6-blocks	0.138 ± 0.230	0.865 ± 0.018	N/A	0.880
	7-blocks	-0.175 ± 0.281	0.844 ± 0.017	N/A	0.860

algorithm learns how to solve the tasks. It is worth noting that the fourth prototype of SD-NLRL generates 65 rules for both STACK and UNSTACK, while it generates 382 rules for ON. On the contrary, NLRL generates 1021 rules for both STACK and UNSTACK, while it generates 2813 rules for ON. Despite SD-NLRL generating more complex rules, the time required to train a SD-NLRL agent is lower than the time required to train a NLRL agent. In fact, NLRL requires approximately 3 days to complete a training process, while SD-NLRL requires approximately 6 hours.

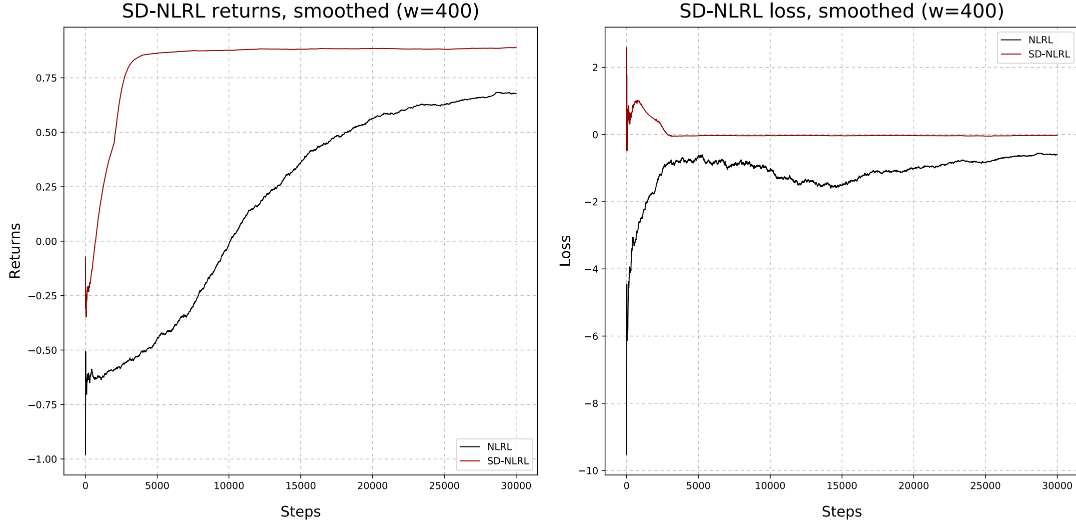

Figure 5.4: Average returns (on the left) and losses (on the right) for the STACK task.

5.8 Discussion

The experimental results suggest that the fourth prototype of SD-NLRL is able to learn to solve a RL task using only the states of the environment and the number of forward chaining steps. In particular, SD-NLRL reaches an overall good performance on the block manipulation tasks, but it performs poorly on the cliff-walking tasks. The cliff-walking tasks require the agent to follow specific paths on the grid, and the sequence of actions required to solve these tasks is longer than the one required by the other tasks. Therefore, it is more difficult to obtain a random positive reward, as documented in [9]. It is worth noting that the rule generation function of SD-NLRL does not try to partially combine the background knowledge with each group. For example, the algorithm could generate a rule for each combination of background atoms that share a constant with the group. However, increasing the number of generated rules implies an additional amount of required resources. It is important to find the best trade-off between the number of generated rules and the required computational resources. Moreover, SD-NLRL is able to capture only a specific chain pattern in the states. This is an example that proves that the capture of recursive pattern within the states can be crucial to improve the performance of a method. However, the algorithm that is presented here is very limited. An interesting research direction would be extending the rule generation function to produce the most valuable and general rules that can solve the task. Moreover, further experiments are needed to understand why SD-NLRL struggles to learn a proper strategy

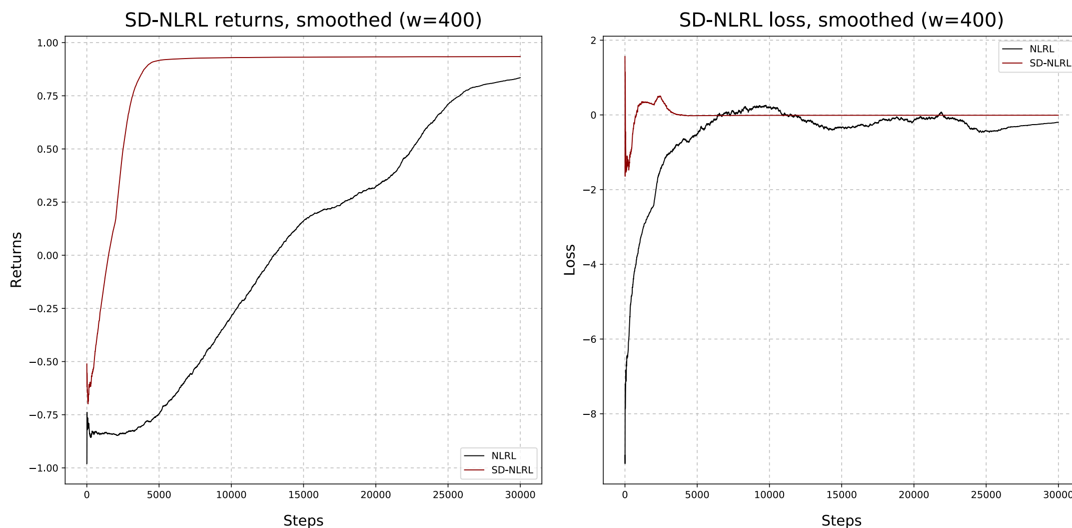

Figure 5.5: Average returns (on the left) and losses (on the right) for UNSTACK.

when the number of constants is increased in the block manipulation tasks.

Another problem of SD-NLRL is that the algorithm is too much sensible to the number of generated rules. When it does not receive a sufficient amount of positive reward, the algorithm quickly learns a wrong strategy, and it stops learning. Moreover, SD-NLRL requires a considerable amount of computational resources, which increase as the number and the complexity of the generated rules increase. Finally, SD-NLRL requires the user to specify a logical representation of all the possible states. These limitations are other interesting challenges for the future. In particular, SD-NLRL could generate new rules only when a new state is encountered, assigning an appropriate weight to each generated rule.

The interpretability of the generated rules is another interesting aspect of SD-NLRL, and the experimental results show that SD-NLRL does not give an appropriate weight to the unreachable rule, as it was initially expected. Therefore, further tests are needed to understand this problem.

The experiments that are discussed in this dissertation are limited because of the limited time and resources. Therefore, it would be interesting to perform additional tests to accurately measure the performance of SD-NLRL. Moreover, the combination of SD-NLRL with a deep neural network would allow the algorithm to work with non-logical input. An interesting research direction would be the measure of the performance of this combination on complex visual tasks, such as the tasks included in OpenAI Gym [27].

SD-NLRL uses REINFORCE as its base RL algorithm. However, the NLRL imple-

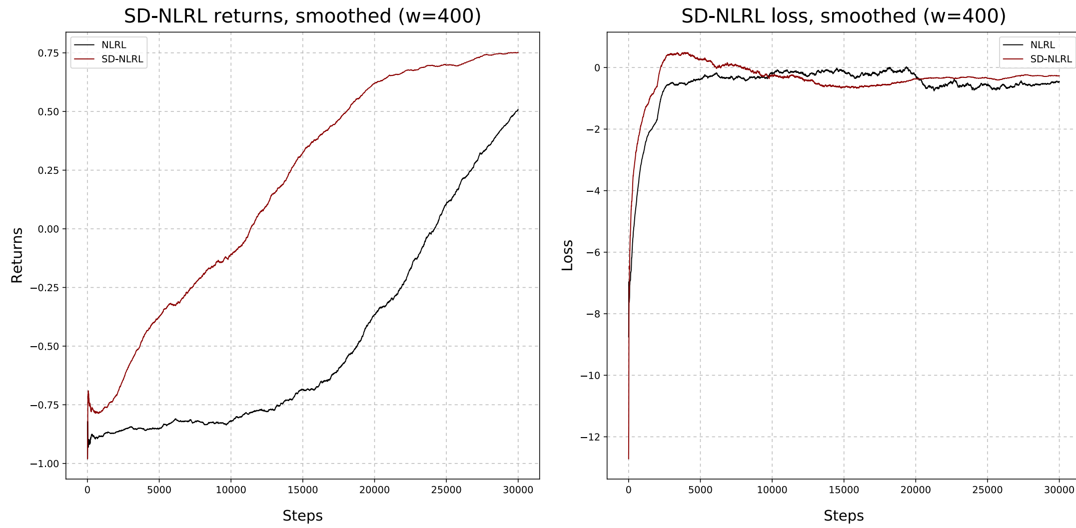

Figure 5.6: Average returns (on the left) and losses (on the right) for the ON task.

mentation includes a modified version of NLRL that is based on PPO [6]. Therefore, another possible extension of the results presented in this dissertation would be combining SD-NLRL with PPO, which is one of the most advanced and used RL method that are currently available.

Conclusions

Deep Reinforcement Learning (*DRL*) and *Relational Reinforcement Learning* (*RRL*) have complementary advantages and limitations. DRL is based on artificial neural networks, which are very difficult to interpret, and the learned policies cannot be easily extracted and reused. Moreover, the generalization capabilities of DRL methods are limited: DRL agents do not effectively solve tasks that are even slightly different from the oned used for training. Therefore, it is difficult to reuse the valuable knowledge that is learned on a simple task to solve complex, but similar, tasks. Finally, DRL methods do not easily use an existing background knowledge. This is often a problem because the agent is forced to learn a large amount of knowledge that could be easily provided by human experts. However, DRL methods are able to cope with uncertain environments, and they can learn directly from non-symbolic input. On the contrary, classic RRL methods are able to generalize to unseen situations, they support knowledge transfer, and they allow the user to specify a structured background knowledge. However, classic RRL methods are not typically able to solve complex tasks, and they are not able to handle the uncertainty of real-world environments.

In order to combine the advantages of symbolic and sub-symbolic *Reinforcement Learning* (*RL*) techniques, neural-symbolic methods for RL have received an increasing interest from the research community in the last few years. This dissertation presented the basic concepts of RL in Section 2. Then, it discussed the most influential DRL methods, as well as the classic RRL methods, in Section 3. DRL methods and classic RRL methods were compared, discussing the main advantages and limitations of the two approaches. The presented discussion suggested that neural-symbolic methods for RL can represent valuable tools to overcome the limitations of both DRL and classic RRL. In fact, neural-symbolic methods for RL combines modern deep learning techniques with a relational representation of states and actions. In addition, this dissertation presented a comparison among the most influential neural-symbolic methods for *Inductive Logic Programming* (*ILP*) that have been introduced in the last five years. In fact, ILP methods can be adapted to solve RL tasks. The neural-symbolic methods for ILP were compared

from the perspective of the interpretability of the learned rules and of the reusabiliy of the learned rules. The comparison suggested that many approaches do not fully support automatic predicate invention. Moreover, ILP methods require the user to specify a large amount of information to learn an appropriate solution to a task, or they imposes too many constraints on rule generation. From a practical point of view, these methods do not offer a complete and well-maintained implementation. Moreover, it is difficult to compare the performance of these methods because a standard set of benchmarks does not exist. Other aspects to consider are the representation of data and the treatment of noisy background knowledge. In fact, only one method is able to learn directly from images, while all other methods require a logic representation of the examples. Actually, none of the compared methods support both infinite domains and noisy background knowledge. Neural-symbolic methods for ILP are more and more studied, and they represent a way to overcome the limitations of traditional ILP.

ILP methods learns logic rules from positive and negative examples. Therefore, they can be adapted to tackle RL tasks. Section 5 introduced a new neural-symbolic method for RL, called *State Driven NLRL* (*SD-NLRL*), which is indirectly based on δILP, a neural- symbolic method for ILP that was discussed in Section 4. Actually, SD-NLRL is based on *Neural Logic Reinforcement Learning* (*NLRL*), which is an adaptation of δILP for RL tasks.

The algorithm proposed in Section 5, namely SD-NLRL, generates candidate rules using a bottom-up approach. In fact, the rules are generated directly from the states of the environment. SD-NLRL uses different techniques to obtain abstract rules from states, and the underlying neural solver learns which is subset of the generated rules that is more appropriate to solve the task. The experimental results showed in Section 5 confirm that SD-NLRL effectively learns to solve different tasks without using program templates. The algorithm requires only the logic representation of all possible states and the number of forward chaining steps. Moreover, SD-NLRL is able to reduce the required time for training and the required computational resources in comparison to NLRL.

Besides its effectiveness in solving the considered learning tasks, there are many problems to be solved before considering SD-NLRL as a definitive tool. SD-NLRL performs poorly on cliff-walking tasks, which are the most difficult among the studied tasks. An important challenge for the future is to extend the current rule generation function to improve the performance of the method on difficult tasks. In particular, it is important to find the best trade-off between the number of generated rules and the required computational resources. The experimental results discussed in Section 5 showed that the capture of the recursive patterns within the states can considerably improve the

performance of the method. However, the current rule generation function is not able to effectively capture general recursive patterns within the states. Moreover, SD-NLRL does not generalize to some block manipulation tasks. The extension of the rule generation function to solve these problems represents another important challenge for the future. SD-NLRL proved to be too much sensible to the number of generated rules, and when this number is too small, the method often remains stuck into a local optimum. Moreover, requiring the user to specify the set of all possible states is a great limitation. Another limitation of SD-NLRL is that the method does not support noisy background knowledge, like many neural-symbolic method for ILP. Moreover, the proposed method does not effectively remove unnecessary rules, and it does not reduce the complexity of the learned policies. Therefore, overcoming these limitations represents other interesting development of this dissertation. Finally, SD-NLRL fails to learn a special rule that can be useful to explicitly avoid actions in specific situations. These are other interesting research directions for the future. The experiments presented in this dissertation are limited, and it would be interesting to measure the performance of SD-NLRL on other tasks. In particular, complex visual tasks represent a great challenge for the future. Moreover, SD-NLRL can be extended by replacing its base RL algorithm, REINFORCE, with modern RL algorithms such as PPO.

In summary, this dissertation presented a comparison among the most influential neural-symbolic methods for ILP that have been introduced in the last five years. Then, it proposed a new neural-symbolic method for RL that showed an overall good performance on five RL tasks. This dissertation also analyzed the major strengths and limitations of both the base algorithm, NLRL, and the proposed algorithm, SD-NLRL, and it discussed several relevant research directions for the future. The proposed algorithm is limited and many improvements are possible. Nonetheless, the experimental results discussed in this dissertation witness that SD-NLRL has the potential to overcome the limitations of the current neural-symbolic methods for RL, allowing to compute interpretable and reusable solutions to complex RL tasks.

Bibliography

[1] R.S. Sutton and A.G. Barto. *Reinforcement Learning, second edition: An introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[2] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. Deep reinforcement learning: An overview. In Yaxin Bi, Supriya Kapoor, and Rahul Bhatia, editors, *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*, pages 426–440. Springer International Publishing, 2018.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *Preprint arXiv:1312.5602*, 2013.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *Preprint arXiv:1707.06347*, 2017.

[7] S. Džeroski, L. De Raedt, and H. Blockeel. *Relational reinforcement learning*, volume 1446 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, 1998.

[8] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[9] Zhengyao Jiang and Shan Luo. Neural logic reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the* 36th *International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3110–3119. PMLR, 09–15 Jun 2019.

[10] Matthieu Zimmer, Xuening Feng, Claire Glanois, Zhaohui Jiang, Jianyi Zhang, Paul Weng, Li Dong, Hao Jianye, and Liu Wulong. Differentiable logic machines. *Preprint arXiv:2102.11529*, 2021.

[11] Ali Payani and Faramarz Fekri. Incorporating relational background knowledge into reinforcement learning via differentiable inductive logic programming. *Preprint arXiv:2003.10386*, 2020.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[13] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The* 33th *International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937. PMLR, 20–22 Jun 2016.

[14] Z. Wang, V. Mnih, V. Bapst, R. Munos, N. Heess, K. Kavukcuoglu, and N. De Freitas. Sample efficient actor-critic with experience replay. In 5th *International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[15] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

[16] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan,

and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 3791–3803. Curran Associates, Inc., 2017.

[17] Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. Towards neural theorem proving at scale. *Preprint arXiv:1807.08204*, 2018.

[18] Pasquale Minervini, Matko Bošnjak, Tim Rocktäschel, Sebastian Riedel, and Edward Grefenstette. Differentiable reasoning on large knowledge bases and natural language. In *Proceedings of the* 34th *AAAI Conference on Artificial Intelligence (AAAI 2020)*, volume 34, pages 5182–5190. AAAI, 2020.

[19] Ali Payani and Faramarz Fekri. Learning algorithms via neural logic networks. *Preprint arXiv:1904.01554*, 2019.

[20] Ali Payani and Faramarz Fekri. Inductive logic programming via differentiable deep neural logic networks. *Preprint arXiv:1906.03523*, 2019.

[21] Andres Campero, Aldo Pareja, Tim Klinger, Josh Tenenbaum, and Sebastian Riedel. Logical rule induction and theory learning using neural theorem proving. *Preprint arXiv:1809.02193*, 2018.

[22] Wang-Zhou Dai and Stephen Muggleton. Abductive knowledge induction from raw data. In Zhi-Hua Zhou, editor, *Proceedings of the* 30th *International Joint Conference on Artificial Intelligence (IJCAI 2021)*, pages 1845–1851. International Joint Conferences on Artificial Intelligence Organization, 2021.

[23] Davide Beretta, Stefania Monica, and Federico Bergenti. Recent neural-symbolic approaches to ILP based on templates. In Davide Calvaresi, Amro Najjar, Michael Winikoff, and Kary Främling, editors, *Proceedings of the* 5th *International Workshop on Explainable and Transparent AI and Multi-Agent Systems*, pages 75–89. Springer International Publishing, 2022.

[24] Davide Beretta, Stefania Monica, and Federico Bergenti. A comparative study of three neural-symbolic approaches to inductive logic programming. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Proceedings of the* 16th *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 56–61. Springer International Publishing, 2022.

[25] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 03 1994.

[26] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[27] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *Preprint arXiv:1606.01540*, 2016.

[28] Peng Peng, Quan Yuan, Ying Wen, Yaodong Yang, Zhenkun Tang, Haitao Long, and Jun Wang. Multiagent bidirectionally-coordinated nets for learning to play starcraft combat games. *Preprint arXiv:1703.10069*, 2, 2017.

[29] Yuxin Wu and Yuandong Tian. Training agent for first-person shooter game with actor-critic curriculum learning. In *Proceedings of the* 5th *International Conference on Learning Representations*, 2017.

[30] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Frcitas. Dueling network architectures for deep reinforcement learning. In 33th *International Conference on Machine Learning, ICML 2016*, volume 4, pages 2939–2947, 2016.

[31] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In 4th *International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[32] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-Learning. *Proceedings of the* 30th *AAAI Conference on Artificial Intelligence*, 30(1), 2016.

[33] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[34] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[35] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In Luc De Raedt and Peter Flach, editors, *Machine Learning: ECML 2001*, pages 97–108. Springer Berlin Heidelberg, 2001.

[36] Kurt Driessens and Jan Ramon. Relational instance based regression for relational reinforcement learning. In *Proceedings of the 20th International Conference on International Conference on Machine Learning*, ICML'03, page 123–130. AAAI Press, 2003.

[37] Thomas Gärtner, Kurt Driessens, and Jan Ramon. Graph kernels and gaussian processes for relational reinforcement learning. In Tamás Horváth and Akihiro Yamamoto, editors, *Inductive Logic Programming*, pages 146–163. Springer Berlin Heidelberg, 2003.

[38] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[39] A. Asperti, D. Cortesi, C. de Pieri, G. Pedrini, and F. Sovrano. Crawling in Rogue's dungeons with deep reinforcement techniques. *IEEE Transactions on Games*, pages 1–1, 2019.

[40] Remi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[41] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32th International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR.

[42] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[43] Luc De Raedt and Hendrik Blockeel. Using logical decision trees for clustering. In Nada Lavrač and Sašo Džeroski, editors, *Inductive Logic Programming*, pages 133–140. Springer Berlin Heidelberg, 1997.

[44] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[45] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of*

the 12th *International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'91, page 726–731. Morgan Kaufmann Publishers Inc., 1991.

[46] David JC MacKay et al. Introduction to gaussian processes. *NATO ASI series F computer and systems sciences*, 168:133–166, 1998.

[47] Gary Marcus. Deep learning: A critical appraisal. *Preprint arXiv:1801.00631*, 2018.

[48] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel. Adversarial attacks on neural network policies. In 5th *International Conference on Learning Representations, ICLR 2017 - Workshop Track Proceedings*, 2017.

[49] Ken Kansky, Tom Silver, David A. Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott Phoenix, and Dileep George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the* 34th *International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1809–1818. PMLR, 06–11 Aug 2017.

[50] Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *Proceedings of the* 41th *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2018)*, pages 0210–0215. IEEE, 2018.

[51] Dongran Yu, Bo Yang, Dayou Liu, and Hui Wang. A survey on neural-symbolic systems. *Preprint arXiv:2111.08164*, 2021.

[52] Roberta Calegari, Giovanni Ciatto, and Andrea Omicini. On the integration of symbolic and sub-symbolic techniques for XAI: A survey. *Intelligenza Artificiale*, 14(1):7–32, 2020.

[53] Andrew Cropper, Sebastijan Dumančić, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Machine Learning*, pages 1–26, 2021.

[54] Luc de Raedt, Sebastijan Dumančić, Robin Manhaeve, and Giuseppe Marra. From statistical relational to neuro-symbolic artificial intelligence. In Christian Bessiere, editor, *Proceedings of the* 29th *International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4943–4950. International Joint Conferences on Artificial Intelligence Organization, 7 2020.

[55] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. Neuro-symbolic artificial intelligence: Current trends. *Preprint arXiv:2105.05330*, 2021.

[56] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[57] Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[58] Yuan Yang and Le Song. Learn to explain efficiently via neural logic inductive learning. In *Proceedings of the* 8th *International Conference on Learning Representations (ICLR 2020)*, 2020.

[59] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in DeepProbLog. *Artificial Intelligence*, 298(C), 2021.

[60] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the* 20th *International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2468–2473. Morgan Kaufmann Publishers Inc., 2007.

[61] David S. Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern, 1988.

[62] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou. Neural logic machines. In 7th *International Conference on Learning Representations, ICLR 2019*, 2019.

[63] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In 5th *International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[64] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of logic and computation*, 2(6):719–770, 1992.

[65] Stephen H. Muggleton, Dianhuan Lin, Jianzhong Chen, and Alireza Tamaddoni-Nezhad. Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors, *Inductive Logic Programming*, pages 1–17. Springer Berlin Heidelberg, 2014.

[66] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In 4th *International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[67] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3):245–286, 1995.

Acknowledgements

I am very grateful to prof. Federico Bergenti for his huge patience and his guidance during this long journey. I would also like to thank the entire AILAB group for its support.
I want to thank my friends Marco Conciatori and Daniele Cortesi for their useful advices and the interesting conversations that helped me when I felt stuck.