Capturing a Recursive Pattern in Neural-Symbolic Reinforcement Learning

Davide Beretta[1], Stefania Monica[2] and Federico Bergenti[1,*]

[1]Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università degli Studi di Parma, Italy
[2]Dipartimento di Scienze e Metodi dell'Ingegneria, Università degli Studi di Modena e Reggio Emilia, Italy

Abstract

Neural-symbolic methods have gained considerable attention in recent years because they are valid approaches to obtain synergistic integration between deep reinforcement learning and symbolic reinforcement learning. Along these lines of research, this paper presents an extension to a recent neural-symbolic method for reinforcement learning. The original method, called State-Driven Neural Logic Reinforcement Learning, generates sets of candidate logic rules from the states of the environment, and it uses a differentiable architecture to select the best subsets of the generated rules that solve the considered training tasks. The proposed extension modifies the rule generation procedure of the original method to effectively capture a recursive pattern among the states of the environment. The experimental results presented in the last part of this paper provide empirical evidence that the proposed approach is beneficial to the learning process. Actually, the proposed extended method is able to tackle diverse tasks while ensuring good generalization capabilities, even in tasks that are problematic for the original method because they exhibit recursive patterns.

Keywords

Machine learning, Reinforcement learning, Neural-symbolic learning

1. Introduction

Neural-symbolic methods for reinforcement learning have been extensively studied in the last few years (e.g., [1]). These methods try to overcome the limitations of deep reinforcement learning methods while preserving their ability to effectively cope with partially observable and stochastic environments. Among the mentioned limitations, it is worth noting that, even if deep reinforcement learning methods obtained notable results on several complex tasks (e.g., [2, 3, 4]), they typically have limited generalization capabilities (e.g., [5, 6]). Moreover, the models that deep reinforcement learning methods produce are not easily interpretable. Neural-symbolic methods for reinforcement learning are expected to overcome both limitations [5].

Three interesting neural-symbolic methods for reinforcement learning are *Differentiable Logic Machine* (*DLM*) [7], *differentiable Neural Logic* (*dNL*) [8], and *Neural Logic Reinforcement Learning* (*NLRL*) [5]. These methods combine differentiable architectures with first order logic, and they can be used to define agents that learn sets of logic rules that represent strategies to effectively

WOA 2023: 24th Workshop From Objects to Agents, November 6–8, Rome, Italy

*Corresponding author.

davide.beretta@unipr.it (D. Beretta); stefania.monica@unimore.it (S. Monica); federico.bergenti@unipr.it (F. Bergenti)

solve the target tasks in environments that present uncertainty. However, the mentioned methods require a large amount of information to guide the search for an effective strategy, which prevents these methods from being used in real-world applications. It is often difficult, or even impossible, to define the structure, or even the characteristics of the final solution for many complex tasks. In addition, these methods often require relevant computational resources even to deal with toy problems.

This paper presents an extension of a recently proposed neural-symbolic method for reinforcement learning, called *State-Driven Neural Logic Reinforcement Learning* (*SD-NLRL*) [9]. The goal of this work is to extend SD-NLRL to capture the recursive patterns among the states of the environment. A recursive pattern can be defined as a set of atoms in which each pair of atoms in the set shares at least a constant, and the set can be replaced with a recursive predicate to obtain a new state which is equivalent to the initial one. In particular, SD-NLRL is based on NLRL, which is an adaptation of ∂ILP [10] for reinforcement learning tasks. In order to solve a specific task, NLRL requires a large amount of information from the user. The user must specify different characteristics of the generated rules using a set of hyper-parameters. On the contrary, SD-NLRL requires information that can be directly obtained from the environment when the set of the possible states of the environment is known. The algorithm that is the base of this work, namely SD-NLRL, generates a set of candidate rules directly from the states of the environment. Then, SD-NLRL employs a modified version of the differentiable architecture of NLRL to select the best subset of the generated rules that are able to solve the training task.

In [9], SD-NLRL is compared with NLRL on five tasks, but SD-NLRL is not able to solve three block manipulation tasks because these tasks present recursive patterns, and SD-NLRL is not able to generate compact and sufficiently general rules from the states. The proposed extension is able to tackle even block manipulation tasks, as shown in Section 3.

SD-NLRL is closely related to another neural-symbolic method for reinforcement learning called dNL. dNL builds a neural network that is used to learn a logical formula in conjunctive or disjunctive normal form. dNL requires the user to specify both the structure of the network and the number of free variables that can be used in the learned rules. Therefore, as discussed for NLRL, SD-NLRL requires less information from the user about the structure of the learned rules, and it does not require the user to size a neural network.

SD-NLRL is also related to *Relational Reinforcement Learning* (*RRL*) [11], which combines Q-learning [12] with a logical representation of states and actions, because they both work on reinforcement learning using logic. In particular, RRL induces a relational regression tree that maps a Q-value, which represents the expected rewards for an action taken in a given state, to each pair of state and action. Several extensions of the original RRL method have been proposed (e.g., [13, 14, 15]). However, these extensions do not exploit deep reinforcement learning and related innovations. On the contrary, SD-NLRL can be used to learn logic rules using a differentiable architecture, which can take advantage of deep reinforcement learning.

Finally, another recent and notable neural-symbolic method for reinforcement learning is DLM, which defines a neural architecture that mimics a logical deduction process. In order to obtain interpretable strategies, DLM can be used to learn ground rules only, and it requires the user to properly size the neural network. On the contrary, SD-NLRL requires the set of all states, which is a strong requirement but, it does not require the user to tune a neural network for the specific purpose of mimicking the deduction process.

The remaining of the paper is organized as follows. Section 2 briefly describes SD-NLRL. Section 2 also reports the experimental results of SD-NLRL on cliff-walking tasks, for completeness. Section 3 presents the proposed extension of SD-NLRL, which is useful when the states of the environment are characterized by recursive patterns. Section 3 also discusses a comparison between the proposed extension of SD-NLRL and NLRL on block manipulation tasks. Finally, Section 4 concludes this paper and outlines possible future developments.

2. Background

This section summarizes SD-NLRL, as described in [9], and it briefly introduces Datalog, which is the language considered throughout this paper. Then, it presents NLRL, which is the neural-symbolic method for reinforcement learning that is the base of SD-NLRL. Then, SD-NLRL is briefly presented, discussing the main differences with NLRL. Finally, the experimental results reported in [9] are presented, for completeness.

2.1. Datalog

SD-NLRL can be used to learn rules that are written in a subset of first-order logic, called Datalog. In Datalog, there are three main primitives: predicate symbols, variable symbols, and constant symbols. Predicate symbols, or predicates, denote relations among objects in the considered domain. Constant symbols, or constants, denote the objects of the considered domain. Variable symbols, or variables, denote references to unspecified objects of the considered domain. An atom γ is defined as $p(t_1, \dots, t_n)$, where p is a predicate and $t_1, \dots, t_n$ are terms, which can be either variables or constants. An atom is called ground when the terms $t_1, \dots, t_n$ are constants. A definite clause is a rule defined as $\gamma \leftarrow \gamma_1, \dots, \gamma_n$, where γ is the head of the rule, and $\gamma_1, \dots, \gamma_n$ is the body of the rule. In this work, the word rule is used as a synonym of definite clause to improve readability. A predicate defined using ground atoms only is called extensional predicate. On the contrary, a predicate defined using a set of definite clauses is called intensional predicate.

2.2. NLRL

NLRL is an adaptation of ∂ILP, a neural-symbolic method for Inductive Logic Programming [16] for reinforcement learning tasks. NLRL requires that both the environment and the background knowledge are presented using ground atoms, and it produces sets of logic rules using program templates, which are sets of hyper-parameters that describe the form of the generated rules. In particular, NLRL generates sets of rules before the training phase, and then it can be used to learn the subset of the generated rules that represent a good strategy to solve the task. In order to perform rule generation, the user must define both the action predicates, representing the possible actions, and the auxiliary predicates, used for predicate invention. During the training phase, NLRL performs a predetermined number of forward chaining steps to obtain the truth value of the action atoms. A program template indicates both the number and the arity of the auxiliary predicates, the number of deduction steps T, and a set of rule templates. A program template must specify a rule template for each intensional predicate, either action or auxiliary. A rule template indicates the number of free variables in the corresponding rule and whether intensional predicates are allowed in the body of the rule. NLRL generates the candidate rules using a top-down approach, and it enforces several constraints on the produced rules. Besides the limitations imposed by the program template, NLRL requires that the body of each rule contains exactly two atoms, and many additional constraints are applied to further reduce the space of generated rules. Therefore, using a program template, NLRL requires the user to specify many details on the form of the final solution, which are not always available for many real-world problems.

In order to select the best subset of rules that solve the training task, NLRL defines a trainable weight for each generated rule. In particular, the algorithm defines a vector of weights θ_i^p for each predicate p and for each rule template i. Moreover, NLRL defines a valuation vector [10] $e \in E = [0,1]^{|G|}$, where G is the set of ground atoms. Each component of e represents how much the algorithm believes that the corresponding atom is true. In order to obtain the truth value of each ground atom, NLRL performs T forward chaining steps, and the sequence of deduction steps for iteration t can be formalized as $f : E \times E \to E$, defined as follows:

$$f^t(e, e_0) = \begin{cases} g(f^{t-1}(e, e_0), e_0) & \text{if } t > 0 \\ e_0 & \text{if } t = 0 \end{cases} \tag{1}$$

where e_0 represents the initial valuation vector, and g describes a single deduction step, which can be formalized as:

$$g(e, e_0) = e_0 + \sum_p \bigoplus_{0 \leq i \leq n} \sum_{0 \leq j \leq k} w_{i,j}^p h_{i,j}^p(e), \tag{2}$$

where n represents the number of rule templates defined for predicate p, k is the number of rules produced using the i-th rule template. $w_{i,j}^p$ is the weight of the j-th rule produced using the i-th rule template for predicate p, and $h_{i,j}^p(e)$ denotes a forward chaining step using the j-th rule produced using the i-th rule template for predicate p. The symbol $\oplus$ denotes the probabilistic sum, which is defined as $a \oplus b = a + b - a \odot b$, where $\odot$ represents the component-wise multiplication. In order to select a single rule for each rule template, NLRL uses a softmax function to obtain each weight $w_{i,j}^p$ from the underlying vector of weights θ_i^p:

$$w_{i,j}^p = \text{softmax}_j(\theta_i^p), \tag{3}$$

where $\text{softmax}_j(x) = \frac{\exp(x_j)}{\sum_l \exp(x_l)}$.

In order to complete the formalization of NLRL, $h_{i,j}^p(e)$ must be defined. Let c be a rule, identified by (p, i, j), the corresponding function $h_{i,j}^p : E \to E$ takes a valuation vector containing the expected truth values of the ground atoms, and it outputs a valuation vector that contains the truth values of the atoms after the application of c. Each function h_c is implemented by a matrix $X_c \in \mathbb{N}^{n \times w \times 2}$, where w denotes the maximum number of pairs of atoms that entail a ground atom. Each X_c is built before the training phase, and it is defined as follows:

$$\begin{gathered} X_c[k, m] = \begin{cases} x_k[m] & \text{if } m < |x_k| \\ (0, 0) & \text{otherwise} \end{cases} \\ x_k = \{(a, b) \mid \text{satisfies}_c(\gamma_a, \gamma_b) \wedge \text{head}_c(\gamma_a, \gamma_b) = \gamma_k\} \end{gathered} \tag{4}$$

where x_k is a set of indexes pairs, and each pair is a reference to a pair of atoms that logically entails the ground atom of index k. Each matrix X_c typically contains many unused indexes pairs $(0, 0)$, which must refer to a pair of atoms. Therefore, the algorithm adds the falsum atom $\perp$ in G, and it associates $(0, 0)$ to $(\perp, \perp)$. During the training phase, NLRL extracts two slices of X_c, namely $X_1, X_2 \in \mathbb{N}^{n \times w}$, such that X_1 and X_2 extract all the indexes of the first atoms from X_c and all the indexes of the second atoms from X_c, respectively:

$$X_1 = X_c[\_, \_, 0] \quad \text{and} \quad X_2 = X_c[\_, \_, 1]. \tag{5}$$

Then, the algorithm obtains the truth value of each ground atom, obtaining $Y_1, Y_2 \in [0, 1]^{n \times w}$, using the function $gather : E \times \mathbb{N}^{n \times w} \to \mathbb{R}^{n \times w}$:

$$Y_1 = gather(e, X_1) \quad \text{and} \quad Y_2 = gather(e, X_2). \tag{6}$$

where *gather*, as defined as in [5], is a function that retrieves the values from a valuation vector given a matrix of indexes. Then, NLRL computes $Z_c \in [0, 1]^{n \times w}$, defined as follows:

$$Z_c = Y_1 \odot Y_2. \tag{7}$$

Finally, the algorithm computes $h_c(e)$ as:

$$h_c(e) = e' \qquad \text{where} \qquad e'[k] = \max_{1 \le j \le w} Z_c[k, j]. \tag{8}$$

It is worth noting that the maximum operator realizes a fuzzy disjunction, while the component-wise multiplication implements a fuzzy conjunction.

2.3. SD-NLRL

SD-NLRL generates the candidate rules directly from the states of the environment. It uses the list of all possible states, which can be provided by the environment in many situations, and the number of forward chaining steps T.

The rule generation function of SD-NLRL, called `generates_rules`, is shown in Algorithm 1. The function has three arguments: the set of action atoms A, the set of states S, and the set of background atoms B. Each state is subdivided into a set of groups V using `get_groups`, and each group includes a subset of the atoms that are included in the state. In particular, each atom in a group is directly or indirectly related with other atoms in the same group through its constants. Therefore, each group has a different set of constants, and it describes a logical characteristic of the state that gives a contribution to the truth value of the action atom. The `generates_rules` function then uses `get_shared` to obtains, for each pair of group and action, the constants that are present both in the action atom and in the group. Then, `generate_rules` adds the background atoms that include at least one shared constant into the group. Here, only the shared constants are considered because they represent a connection between the groups of the state and the action to be performed. The function then obtains the set of unshared constants, which are the constants that are not shared between the action atom and the group, after the related background atoms are included. Now, when the unshared constants are not present, the final rule is produced using `unground`, which converts each ground term into a

Algorithm 1 The function that performs rule generation in SD-NLRL.

```
function GENERATE_RULES($A, S, B$)
    rules ← {}
    for each $s \in S$ do
        $V$ ← get_groups($s$)
        for each $g \in V$ do
            for each $a \in A$ do
                shared ← get_shared($a, g$)
                insert atoms $b \in B$ into $g$ such that consts($b$) $\cap$ shared $\neq \emptyset$
                unshared ← get_unshared($a, g$)
                if unshared $= \emptyset$ then
                    $r$ ← unground($a, g$, {})
                    rules ← rules $\cup$ $r$
                else
                    for each $f \in$ unshared do
                        fixed ← unshared $\setminus \{f\}$
                        partial_rule ← unground($a, g$, fixed)
                        add $b \in B$ to the body of partial_rule such that $f \in$ consts($b$)
                        r ← unground($a$, body(partial_rule), {})
                        rules ← rules $\cup$ r
                    end for
                end if
            end for
        end for
    end for
    return rules
end function
```

new variable. Otherwise, a rule is generated for each unshared constant *f*. In particular, the set of unshared constants that does not include *f* is immediately transformed into variables. Then, the function adds the background atoms that include *f* to the body of the rule. Finally, the remaining constants are converted into variables.

The generation of a new rule for each unshared constant is useful to decrease the complexity of each rule. In fact, the algorithm tries to obtain the most simple characteristics from each state. Then, the differentiable architecture picks the best subset of rules that solves the training task. When converting constants into variables, `unground` starts from the body, which is the second argument, and it transforms the rule going from left to right. Finally, it changes the head of the rule, which is the first argument. The last argument is a set of constants, and the function converts only the constants in the set when the latter is not empty.

SD-NLRL allows an action predicate to be described by an unpredetermined number of rules. Moreover, there is not limit on both the arity of the predicates and the number of atoms that are included in the body of the rules. As a consequence, SD-NLRL defines an architecture that is different from the one used by NLRL, and it defines a single deduction step g as:

$$g(e, e_o) = e_o + \sum_{p} \bigoplus_{0 \leq j \leq k} w_j^p h_j^p(e), \tag{9}$$

where w_j^p is the weight of rule j for predicate p, and $h_j^p(e)$ denotes a forward chaining step using the rule j defined for predicate p. The values of weights w_j^p are restricted in $[0, 1]$ applying the following normalization function:

$$w_j^p = \frac{\max(w^p) - w_j^p}{\max(w^p) - \min(w^p)}. \tag{10}$$

In order to implement $h_c(e)$ for rule c, SD-NLRL defines a matrix $X_c \in \mathbb{N}^{n \times w \times d}$ for each rule c:

$$X_c[k, m] = \begin{cases} x_k[m] & \text{if } m < |x_k| \\ (0, \ldots, 0) & \text{otherwise} \end{cases}$$

$$x_k = \{(a_1, \ldots, a_d) \mid \text{satisfies}_c(\gamma_{a_1}, \ldots, \gamma_{a_d}) \wedge \text{head}_c(\gamma_{a_1}, \ldots, \gamma_{a_d}) = \gamma_k\}$$

Then, SD-NLRL computes matrices $X_1, \ldots, X_d \in \mathbb{N}^{n \times w}$ before starting the training process. The algorithm then computes $Y_1, \ldots, Y_d \in \mathbb{R}^{n \times w}$ using *gather*, and it obtains Z_c, defined as:

$$Z_c = Y_1 \odot \cdots \odot Y_d. \tag{11}$$

Finally, the algorithm defines h_c using (8).

SD-NLRL is able to associate the same value to different rule weights. Note that the proposed normalization (10) has some problems. In fact, it is not applicable if $\max(w^p) = \min(w^p)$, e.g., a single weight. Moreover, using this normalization imposes that both a rule with weight 0 and a rule with weight 1 must always be present. These problems can be detrimental in some situations, although for most real-world applications these problems should not be present.

In [9], the performance of SD-NLRL has been evaluated on the same reinforcement learning tasks that are discussed in the paper that originally introduced NLRL [5]. In particular, SD-NLRL has been evaluated on two cliff-walking tasks and three block manipulation tasks. The cliff-walking tasks are CLIFFWALKING and WINDYCLIFFWALKING, where the agent must move from an initial position to a goal position on a 5x5 grid without going to a cliff position and using as few moves as possible. The difference between CLIFFWALKING and WINDYCLIFFWALKING is that in the second task the agent has a 10% chance of moving downward on the grid, independently from the taken action.

In [9], the block manipulation tasks are STACK, UNSTACK, and ON. STACK requires the agent to build a single column of blocks, UNSTACK requires the agent to put all the blocks on the floor, while in ON the agent must put a specific block onto another specific block. The evaluation of the learned strategies was made using the same variations of the tasks described in [5]. In particular, many variations of the same task are used to evaluate the generalization capabilities of the method. SD-NLRL algorithm fails to solve block manipulation tasks [9] because it is not possible to subdivide states into small groups of atoms. In particular, in block manipulation tasks the atoms of the states are connected to each other because all the blocks

Table 1
A performance comparison between NLRL and SD-NLRL on the CLIFFWALKING and WINDYCLIFFWALKING tasks. The table is taken from [9], and the optimal returns (Optimal column) are taken from the paper that introduces NLRL [5].

Environment	Task	NLRL	SD-NLRL	Optimal
CLIFFWALKING	train	0.862 ± 0.026	0.674 ± 0.292	0.880
	top-left	0.749 ± 0.057	0.652 ± 0.200	0.840
	top-right	0.809 ± 0.064	0.781 ± 0.038	0.920
	centre	0.859 ± 0.050	0.775 ± 0.198	0.920
	6x6	0.841 ± 0.024	0.631 ± 0.381	0.860
	7x7	0.824 ± 0.024	0.520 ± 0.514	0.840
WINDYCLIFFWALKING	train	0.663 ± 0.377	−0.808 ± 0.341	0.769 ± 0.162
	top-left	0.726 ± 0.075	−0.536 ± 0.480	0.837 ± 0.068
	top-right	0.834 ± 0.061	−0.290 ± 0.548	0.920 ± 0.000
	centre	0.672 ± 0.579	−0.350 ± 0.430	0.868 ± 0.303
	6x6	0.345 ± 0.736	−0.991 ± 0.093	0.748 ± 0.135
	7x7	0.506 ± 0.528	−1.012 ± 0.047	0.716 ± 0.181

are placed upon other blocks or they are on the floor. Therefore, they are connected to the other blocks directly through the `on/2` predicate, or indirectly through the floor constant, which is common to every pile of block. For example, the following state represents a pile of blocks:

```
top(a), on(a,b), on(b,c), on(c,floor)
```

in which block `c` is on the floor, block `b` is on block `c`, and block `a` is on block `b` at the top of the pile. This state produces a single group of atoms. This is a characteristic of the states of the considered block manipulation tasks. Therefore, SD-NLRL produces complex rules, and the learning process becomes more computationally demanding as both the number and the size of the generated rules increase.

Table 1, taken from [9], reports the average returns of SD-NLRL. In order to obtain the results, 5 runs have been made for each training task, and the resulting models have been evaluated on 100 episodes for each task. Finally, the returns have been averaged over all runs for each task. The variants of CLIFFWALKING and WINDYCLIFFWALKING, which are used to measure the generalization capabilities of the algorithm, are 5: *top-left*, *top-right*, and *center* impose a different starting position of the agent, while *6x6* and *7x7* increase the size of the grid. From the results shown in Table 1 emerges that SD-NLRL is able to effectively solve CLIFFWALKING, and it is able to generalize to unseen states. However, the results show great variance, and SD-NLRL is not able to learn an effective strategy for WINDYCLIFFWALKING. The agent often remains trapped into a local optimum. This behaviour is justified by the rule generation strategy, which produces rules from the states of the environment, and it is not able to generate the best rules for each action predicate.

3. Handling a recursive pattern

One of the main issues of SD-NLRL is that it is not able to handle deeply connected states because compact rules cannot be generated. This prevented SD-NLRL from generating an effective strategy for block manipulation tasks. This section presents the proposed extension of SD-NLRL to deal with a form of deeply connected states. In particular, the considered states exhibit recursive patterns. The proposed extension can be easily adapted to capture other recursive patterns within the states. Note that the proposed algorithm behaves as the original one on cliff-walking tasks because the states of the cliff-walking tasks do not contain recursive patterns. Therefore, the results shown in Table 1 are still valid for the proposed extension.

As previously discussed, the states of block manipulation tasks discussed in [5] follow a specific pattern. For example, a pile of blocks is always defined by:

$$\texttt{top}(x_1),\ \texttt{on}(x_1,x_2),\ \texttt{on}(x_2,x_3),\ \ldots,\ \texttt{on}(x_N,\texttt{floor}),$$

where N is a number and `on(X,Y)` is a predicate which specifies that a block X is on Y, where Y is either a block or the floor. The other predicate, `top(X)`, specifies that block X is the top block of a pile. In order to reduce the size and the number of the generated rules, it is important to capture the recursive patterns within states.

The discussed block manipulation tasks require the agent to capture the concept of a pile of blocks, which can be modeled as:

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

The predicate `pile/2` is able to capture the chain relation among the blocks. The proposed extension of SD-NLRL defines the auxiliary predicate `pile/2` during the rule generation phase, and it replaces the occurrences of `on/2` in the generated rules with `pile`(A,B), where A and B are the constants that limit the sequence on the left and on the right, respectively. An example of the application of `replace_chain` is provided:

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

is rewritten as a more compact set of atoms:

```
top(a), pile(a,floor), top(d), on(d,floor).
```

In particular, Algorithm 1 is modified adding two lines, as shown in Algorithm 2. The function `replace_chain` performs the substitution of the connected predicates, and it applies a sorting algorithm to the group of atoms. The function sorts the atoms looking at the names of the predicates. When two predicates have the same name, the function uses the arguments, proceeding from left to right. The ordering on the atoms is enforced to reduce the number of generated rules after the call to `unground`. For example, starting from the following groups of atoms, which both define a pile of blocks and a single block on the floor:

```
top(a), pile(a,floor), top(d), on(d,floor),
top(a), top(d), pile(a,floor), on(d,floor),
```

Algorithm 2 The proposed modification to the rule generation function, the new lines are marked with frames.

```
function GENERATE_RULES($A, S, B$)
    rules $\leftarrow$ {}
    for each $s \in S$ do
        $V \leftarrow$ get_groups($s$)
        for each $g \in V$ do
            for each $a \in A$ do
                shared $\leftarrow$ get_shared($a, g$)
                insert atoms $b \in B$ into $g$ such that consts($b$) $\cap$ shared $\neq \emptyset$
                unshared $\leftarrow$ get_unshared($a, g$)
                if unshared $= \emptyset$ then
                    $\boxed{g \leftarrow \text{replace\_chain}(g)}$
                    $r \leftarrow$ unground($a, g, \{\}$)
                    rules $\leftarrow$ rules $\cup\ r$
                else
                    for each $f \in$ unshared do
                        fixed $\leftarrow$ unshared $\setminus \{f\}$
                        partial_rule $\leftarrow$ unground($a, g$, fixed)
                        add $b \in B$ to the body of partial_rule such that $f \in$ consts($b$)
                        $\boxed{\text{body(partial\_rule)} \leftarrow \text{replace\_chain(body(partial\_rule))}}$
                        r $\leftarrow$ unground($a$, body(partial_rule), {})
                        rules $\leftarrow$ rules $\cup$ r
                    end for
                end if
            end for
        end for
    end for
    return rules
end function
```

and considering the action atom `move(a,d)`, if the ordering is not enforced on the body of the rule the method generates the following rules:

```
move(X,Z) :- top(X), pile(X,Y), top(Z), on(Z,Y).
move(X,Y) :- top(X), top(Y), pile(X,Z), on(Y,Z).
```

Therefore, the algorithm would generate two rules that are equivalent. On the contrary, after ordering the atoms and applying `unground`, the body of the generated rules would be:

```
move(Z,X) :- on(X,Y), pile(Z,Y), top(Z), top(X).
move(Z,X) :- on(X,Y), pile(Z,Y), top(Z), top(X).
```

Therefore, the algorithm generates two equal rules, and one rule can be discarded.

Table 2
A performance comparison between NLRL and the proposed extension on the STACK, UNSTACK, and ON tasks. The optimal returns (Optimal column) are taken from the paper that introduces NLRL [5].

Environment	Task	NLRL	SD-NLRL	Optimal
UNSTACK	train	0.937 ± 0.008	0.932 ± 0.002	0.940
	swap-2	0.936 ± 0.009	0.935 ± 0.002	0.940
	divide-2	0.958 ± 0.006	0.956 ± 0.002	0.960
	5-blocks	0.915 ± 0.010	0.906 ± 0.005	0.920
	6-blocks	0.891 ± 0.014	0.873 ± 0.009	0.900
	7-blocks	0.868 ± 0.016	0.835 ± 0.012	0.880
STACK	train	0.910 ± 0.033	0.896 ± 0.056	0.940
	swap-2	0.913 ± 0.029	0.899 ± 0.050	0.940
	divide-2	0.897 ± 0.064	0.882 ± 0.046	0.940
	5-blocks	0.891 ± 0.032	0.840 ± 0.088	0.920
	6-blocks	0.856 ± 0.169	0.755 ± 0.122	0.900
	7-blocks	0.828 ± 0.179	0.508 ± 0.170	0.880
ON	train	0.915 ± 0.010	0.816 ± 0.013	0.920
	swap-2	0.912 ± 0.013	0.809 ± 0.028	0.920
	swap-middle	0.914 ± 0.011	0.734 ± 0.021	0.920
	5-blocks	0.890 ± 0.016	N/A	0.900
	6-blocks	0.865 ± 0.018	N/A	0.880
	7-blocks	0.844 ± 0.017	N/A	0.860

Replacing low-level concepts with more abstract concepts allows the proposed extension to reduce both the number and the size of the generated rules. Therefore, the required computational resources are heavily reduced, and the algorithm is consequently able to complete the training process for block manipulation tasks. Moreover, the agent is able to generalize to piles of blocks of unpredetermined length.

The proposed extension changes also the definition of `get_shared`, which now returns the set of shared constants among all the atoms in the group together with the action atom. The function `get_unshared` has been changed as well. This modification has been made to further reduce the number of generated rules, although their length can grow because the number of shared constants is bigger. In fact, the algorithm may add more background knowledge to each rule. In general, it is not clear which definition should be preferred, and this can represent an interesting research direction for the future.

Table 2 shows the performance of the proposed extension for block manipulation tasks. The results have been obtained following the same procedure that is discussed in the previous section. 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.

Both SD-NLRL and the proposed extension are based on the official implementation of NLRL, which is available at github.com/ZhengyaoJiang/NLRL, and they all share the same hyper-parameters. However, the learning rate and T are specific to each task. In particular, T was set to 4 for block manipulation tasks during the training phase. In the evaluation phase, T was set to 7 because more forward chaining steps are needed to obtain the correct truth value of the action predicates. In fact, the `pile/2` predicate presents a recursive definition, and the number of deduction steps must be greater than the number of recursive calls. The learning rate was set to 0.05 for all tasks.

As shown in Table 2, the obtained results of both NLRL and SD-NLRL with the proposed extension are comparable on both STACK and UNSTACK, but NLRL shows an overall better performance. Moreover, SD-NLRL with the proposed extension shows worse generalization capabilities than NLRL on STACK. Finally, the results on the ON task shows that NLRL outperforms the presented algorithm by a considerable margin, and SD-NLRL with the proposed extension does not effectively generalize to slightly different tasks. In fact, the proposed extension fails to complete the evaluation process on three variants of ON because the realized computational graph exceeds the technical limitations of the implementation. In the case of STACK, the rules that are learned using SD-NLRL suggest that the rule generation process fails to produce a general rule for the initial state, where all the blocks are on the floor. Therefore, the algorithm performs worse as the number of blocks increases because more random moves are needed to build the first pile of blocks. In the case of ON, the task is clearly more difficult than the other tasks, and both the number and size of generated rules increase because an additional predicate, `goal(X,Y)`, is required to specify that block X must be placed on block Y. Therefore, it is even more difficult to generate good candidate rules from states. It is worth noting that, in all cases, only the rules with a weight greater than 0.3 are reported, for space limitations. In summary, the presented experimental results provide empirical evidence that the proposed extension to SD-NLRL is beneficial, even if a much wider experimental campaign is needed to assess convincing and reliable results.

The following example shows another application of this technique, which could be a task where a state of the environment describes a sequence of rectangles and circles. In particular, let `rect(X,Y)` and `circle(X,Y)` represent two type of cells that connect positions X and Y on a path, respectively. Then, the `path(X,Y)` predicate can be used to define a specific path structure, represented as a sequence of rectangles and circles:

```
path(X,Y) :- rect(X,Z), circle(Z,K), rect(K,Y).
path(X,Y) :- circle(X,Z), path(Z,K), circle(K,Y).
```

Note that this technique can be extended to every recursive pattern within the states, even with predicates with arity higher than 2 and different forms of recursion. However, the presented method is not able to automatically detect recursion within states and it does not automatically define the recursive predicate.

In order to provide a comprehensive view on the proposed method, the following part reports the learned rules of the best model for each block manipulation task. In particular, the learned rule (on the right) and the corresponding weight (on the left) for STACK are:

```
1.0     move(Y,M) :- floor(X),on(Y,X),on(Z,X),pile(M,X),top(Y),
                     top(M),top(Z).
```

The learned rule moves a block that is on the floor onto a pile of at least two blocks. Therefore, SD-NLRL can be used to learn a compact strategy that effectively solves the task, but the learned strategy is not the best one because some atoms, namely on(Z,X) and top(Z), may be omitted. The learned rules for UNSTACK are instead:

```
0.63    move(Y,X) :- floor(X),pile(Y,X),pile(Z,X),top(Y),top(Z).
0.63    move(Z,X) :- floor(X),pile(Y,X),pile(Z,X),top(Y),top(Z).
0.63    move(Y,X) :- floor(X),pile(Y,X),top(Y).
0.36    move(Z,X) :- floor(X),on(Y,X),pile(Z,X),top(Z),top(Y).
1.0     move(M,X) :- floor(X),on(Y,X),on(Z,X),pile(M,X),top(Y),
                     top(M),top(Z).
```

In this case, SD-NLRL can be used to learn an effective strategy, but the number of learned rules is even bigger. In fact, the presented strategy is to move a top block on the floor when there is a single pile of blocks, two different piles of blocks or a pile of blocks and one or two blocks on the floor. It is evident that the learned strategy is redundant, and the third rule would be sufficient to solve the task. Finally, the learned rules for ON are:

```
1.0     move(M,X) :- floor(X),goal_on(Y,Z),pile(M,X),pile(N,X),
                     top(M),top(N).
0.64    move(Y,Z) :- floor(X),goal_on(Y,Z),on(Y,X),on(Z,X),
                     on(M,X),on(N,X),top(Y),top(Z),top(M),top(N).
0.53    move(Y,Z) :- floor(X),goal_on(Y,Z),on(Y,X),on(M,X),
                     pile(Z,X),top(Y),top(Z),top(M).
0.55    move(Y,Z) :- floor(X),goal_on(Y,Z),on(Y,X),pile(Z,X),
                     top(Y),top(Z).
```

The proposed method can be used to learn stacking the goal blocks when they are top blocks. When it is not possible to directly stack the goal blocks, SD-NLRL can be used to learn to put a top block on the floor. This strategy is effective but, similarly to the previously discussed strategies, the number of learned rules is excessive. In particular, the third rule should be omitted, and the strategy lacks a rule that moves the top block of a pile onto a block on the floor when the two blocks are the goal blocks. Moreover, the learned rules contain many unnecessary atoms, e.g. pile(N,X) and top(N) in the first rule. Another example is the second rule that has been learned for the ON task, which requires that 4 blocks are on the floor. The learned rule is not only redundant, but also dangerous because it cannot be used in an environment with only 3 blocks. Both these examples show that the learned rules are too much specific with respect to the states of the training environment. In fact, the training environment defines 4 blocks, and the generated rules often represent specific situations only. In summary, from this analysis emerges that, despite being able to learn an effective strategy, many rules that are learned using SD-NLRL are unnecessary or overly specific. In fact, the agent learns a strategy that is specific to the training environment. Therefore, generating more compact and general, yet useful, rules is another important research direction for the future.

4. Discussion

This paper proposed an extension of SD-NLRL, which is a neural-symbolic method for reinforcement learning. The proposed extension modifies the rule generation process of the original algorithm to capture recursive patterns within the states of the environment. The experimental results show that the proposed extension can be used to learn an effective strategy for block manipulation tasks, which are problematic for the original method. Moreover, the algorithm effectively generalizes to unforeseen situations. In particular, when the states exhibit a recursive pattern, the proposed method can be useful to reduce the number of generated rules and the required computational resources. Moreover, SD-NLRL performs worse than NLRL but in many tasks the difference between the algorithms is small in terms of performance, and SD-NLRL solves the tasks using the states of the environment. On the contrary, NLRL requires the user to specify the characteristics of the generated rules. Both are strong requirements but, in many cases, the states of the environment can be automatically obtained from the problem definition, while it can be difficult to manually tune the program templates for complex tasks. Note that the proposed extension could be easily generalized to other tasks.

Despite consistently solving the considered tasks, the proposed extension performs worse than NLRL, and it presents several limitations. In particular, it does not generate sufficiently general and abstract rules, and this affects also the generalization capabilities of the method. Moreover, the proposed extension captures only a specific pattern among states and it is not able to automatically detect recursive patterns within states. The proposed method is also not able to tackle complex problems because it builds large models and it requires a large amount of computational resources to solve simple tasks. Moreover, the experimental results suggest that the proposed extension can be used to learn rules that are either redundant or too much case-specific. Finally, many limitations of the original SD-NLRL algorithm are still present. In fact, the proposed extension still needs the set of possible states, and it sometimes remains trapped into a local optimum during training.

There are many interesting research directions for the future. For example, more effort is needed to generate abstract and general rules from deeply connected states, and the presented approach could be extended to automatically detect different type of recursive patterns within the states. Moreover, the proposed method can be improved by generating the rules iteratively as the agent visit new states. This prevents the environment from specifying the set of all states, and it is useful to reduce the number of generated rules, allowing the method to tackle more complex tasks. Finally, the learning stability should be improved, and more effort is needed to prevent the agent from learning redundant rules. In summary, despite covering a specific case, this work provides convincing experimental evidence of the validity of the approach, which can be a viable means towards successful neural-symbolic reinforcement learning.

References

[1] K. Acharya, W. Raza, C. Dourado, A. Velasquez, H. H. Song, Neurosymbolic reinforcement learning and planning: A survey, IEEE Transactions on Artificial Intelligence (2023) 1–14.

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

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint:1707.06347 (2017).

[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, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (2016) 484–489.

[5] Z. Jiang, S. Luo, Neural logic reinforcement learning, in: Proceedings of the 36th International Conference on Machine Learning, volume 97 of *Proceedings of Machine Learning Research*, 2019, pp. 3110–3119.

[6] G. Marcus, Deep learning: A critical appraisal, arXiv preprint:1801.00631 (2018).

[7] M. Zimmer, X. Feng, C. Glanois, Z. Jiang, J. Zhang, P. Weng, L. Dong, H. Jianye, L. Wulong, Differentiable logic machines, arXiv preprint:2102.11529 (2021).

[8] A. Payani, F. Fekri, Incorporating relational background knowledge into reinforcement learning via differentiable inductive logic programming, arXiv preprint:2003.10386 (2020).

[9] D. Beretta, S. Monica, F. Bergenti, Preliminary results on a state-driven method for rule construction in neural-symbolic reinforcement learning, in: Proceedings of the 17th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy 2023), CEUR Workshop Proceedings, CEUR-WS.org, 2023, pp. 128–138.

[10] R. Evans, E. Grefenstette, Learning explanatory rules from noisy data, Journal of Artificial Intelligence Research 61 (2018) 1–64.

[11] S. Džeroski, L. De Raedt, K. Driessens, Relational reinforcement learning, Machine learning 43 (2001) 7–52.

[12] C. J. Watkins, P. Dayan, Q-learning, Machine learning 8 (1992) 279–292.

[13] K. Driessens, J. Ramon, H. Blockeel, Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner, in: Proceedings of the 12th European Conference on Machine Learning (ECML 2001), Springer, 2001, pp. 97–108.

[14] K. Driessens, J. Ramon, Relational instance based regression for relational reinforcement learning, in: Proceedings of the 20th International Conference on Machine Learning (ICML 2003), AAAI Press, 2003, pp. 123–130.

[15] T. Gärtner, K. Driessens, J. Ramon, Graph kernels and Gaussian processes for relational reinforcement learning, in: Proceedings of the 13th International Conference on Inductive Logic Programming (ILP 2003), Springer, 2003, pp. 146–163.

[16] S. Muggleton, Inductive logic programming, New generation computing 8 (1991) 295–318.