Preliminary Results on a State-Driven Method for Rule Construction 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

Deep reinforcement learning has obtained impressive results in the last few years. However, the limitations of deep reinforcement learning with respect to interpretability and generalization have been clearly identified and discussed. In order to overcome these limitations, neural-symbolic methods for reinforcement learning have been recently proposed. This paper presents preliminary results on a new neural-symbolic method for reinforcement learning called State-Driven Neural Logic Reinforcement Learning. The discussed method generates sets of candidate logic rules directly from the states of the environment. Then, it uses a differentiable architecture to select a good subset of the generated rules to successfully complete the training task. The experimental results presented in this paper provide empirical evidence that the discussed method can achieve good performance without requiring the user to specify the structure of the generated rules. Besides being preliminary, the experimental results also suggest that the presented method has sufficient generalization capabilities to allow using learned rules in environments that are sufficiently similar to the training environment. However, this is a preliminary work, and the experimental results show that the proposed method is not yet sufficiently effective.

Keywords

Artificial intelligence, Machine learning, Reinforcement learning, Neural-symbolic learning

1. Introduction

Deep reinforcement learning has received increasing interest from the research community in recent years because its methods have obtained impressive results on many complex tasks (e.g., [1, 2, 3, 4, 5]). However, the limitations of deep reinforcement learning from the perspective of the interpretability of learned strategies are well known. Moreover, these methods are expected to have limited generalization capabilities because they are unable to reuse learned strategies on tasks that are different from the ones used for training [6, 7].

In order to overcome these two major limitations, neural-symbolic methods for reinforcement learning have gained relevant interest and appreciation. Many research proposals have tried to combine neural approaches with first-order logic, such as DLM [8], dNL [9], and NLRL [7].

NeSy 2023, 17th International Workshop on Neural-Symbolic Learning and Reasoning, Certosa di Pontignano, Siena, Italy

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

These three methods use differentiable architectures [10] to learn sets of logic rules that solve the tasks used for training. Therefore, these methods are able to effectively combine the benefits of neural architectures, such as the ability to cope with uncertain environments, with good interpretability and generalization capabilities. However, these methods require the user to provide large amounts of data to guide the search. Moreover, they have not been currently tested on complex reinforcement learning tasks. Finally, they often require large amounts of computational resources to successfully solve even simple tasks.

This paper proposes a novel neural-symbolic method for reinforcement learning, called *SD-NLRL* (*State-Driven Neural Logic Reinforcement Learning*), that is largely based on NLRL. For a given learning task, NLRL generates a set of candidate rules using a top-down approach that requires the user to specify a set of hyper-parameters, called program template, to guide the generation of rules. On the contrary, SD-NLRL generates its candidate rules directly from the states of the environment. Actually, SD-NLRL uses a modified version of the differentiable architecture of NLRL to select the best subset of the generated rules that solves the task under investigation. In order to fairly compare SD-NLRL with NLRL, SD-NLRL was used on the tasks that are described in the paper [7] that introduced NLRL. Actually, the two methods are compared using two cliff-walking tasks and three block manipulation tasks.

The work presented in this paper is closely related to relational reinforcement learning [11], which represents states and actions using first-order logic and induces a relational regression tree that associates each state-action pair to a Q-value. The literature documents several extensions (e.g., [12, 13, 14]) of the original relational reinforcement learning method. However, these extensions do not use differentiable architectures, and therefore they are not able to exploit the most recent breakthroughs of deep reinforcement learning.

A recent work on neural-symbolic reinforcement learning that is closely related to SD-NLRL is documented in [9], which proposes an adaptation of dNL [15] for reinforcement learning tasks. dNL uses neural networks that learn logic formulae in conjunctive or disjunctive normal forms. dNL requires the user to provide the structure of the neural network together with the number of free variables to be used in the learned rules. On the contrary, SD-NLRL uses a differentiable architecture based on the states of the environment, and it does not require the user to provide additional parameters to define the structure of the learned rules.

Another method that is closely related to SD-NLRL is DLM [8], which is based on NLM [16]. NLM obtained notable results on some reinforcement learning tasks using a deep neural network to mimic the logical deduction. However, the learned policies that NLM provides are not directly interpretable. DLM extends NLM by replacing neural operators with fuzzy operators, and therefore it is able to learn interpretable policies. However, DLM learns ground rules only. Moreover, both NLM and DLM require the user to specify the architecture of the neural network, which is not required by SD-NLRL.

The remaining of this paper is structured as follows. Section 2 discusses the proposed method. Section 3 presents an experimental comparison of SD-NLRL and NLRL. Finally, Section 4 concludes this paper discussing some of the current limitations of SD-NLRL and outlining possible research directions for the future.

2. SD-NLRL

The first part of this section presents a brief recall of the subset of first-order logic, namely Datalog, that is considered in this work. Then, this section proceeds with a brief introduction to NLRL, which is the neural-symbolic method for reinforcement learning that represents the base of SD-NLRL. Finally, SD-NLRL is described focusing on its main differences from NLRL.

2.1. Datalog

Datalog is a subset of first-order logic, and its syntax comprises three main primitives: predicate symbols, variable symbols, and constant symbols. Predicate symbols, or predicates, represent relations among objects in the domain of discourse. Constant symbols, or constants, represent the objects of the domain of discourse. Variable symbols, or variables, represent references to unspecified objects of the domain of discourse. An atom β 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. For example, if *on(X, table)* is an atom that is true if the first object is on the second object, then *on* is the predicate symbol, X is a variable, and *table* is a constant. When the terms $t_1, \dots, t_n$ are all constants, the atom is called ground. A rule in the form $\beta \leftarrow \beta_1, \dots, \beta_n$ is called definite clause. For the sake of simplicity, in this work, the word rule is considered as a synonym of definite clause. When a predicate is defined using ground atoms only, it is called extensional predicate. On the contrary, when a predicate is defined using a set of definite clauses, it is called intensional predicate.

2.2. NLRL

NLRL is an adaptation for reinforcement learning tasks of ∂ILP [10], a neural-symbolic method for ILP. In particular, in NLRL, both the environment and the background knowledge are described using ground atoms, and, for a given task, NLRL generates a set of candidate rules using a set of hyper-parameters called program template. Formally, a program template is defined as a quadruple:

$$\langle \text{Pred}_a, \text{arity}_a, \Pi, T \rangle, \tag{1}$$

where Pred_a and arity_a denote, respectively, the number and the arity of the auxiliary predicates, T denotes the number of forward chaining steps, and Π denotes a set of rule templates for each intensional predicate, either target or auxiliary. Each rule template is formally defined as $\langle v, int \rangle$, where v denotes the number of free variables in the rule, and *int* indicates if the body of the rule can contain intensional predicates. Therefore, NLRL is based on strong assumptions on the rule generation process, and the user must specify many hyper-parameters to guide this process. Moreover, in NLRL, the body of each rule must contain exactly two atoms, and several additional constraints are enforced to further limit the space of generated rules. The needed program template provides many details on the form of the generated rules, and it is often difficult to appropriately tune all these hyper-parameters in real-world applications. It is worth noting that the program template must be accurately designed. In fact, the size of the generated rules grows very quickly with the complexity of the program template, which is determined, for example, by the number of auxiliary predicates.

In order to solve a reinforcement learning task, NLRL associates a trainable weight with each generated rule. In particular, NLRL defines a valuation vector in $E = [0,1]^{|G|}$, where G is the set of ground atoms. Each component of a valuation vector represents how likely the related ground atom is expected to be true. The deductive process can be formalized as a mapping $f : E \times E \to E$ defined, for the t-th iteration, as:

$$f^t(e, e_0) = \begin{cases} g(f^{t-1}(e, e_0), e_0) & \text{if } t > 0 \\ e_0 & \text{if } t = 0, \end{cases} \tag{2}$$

where e_0 is the initial valuation vector, and g represents a single forward chaining step. Let $\oplus$ denote the probabilistic sum, which is defined as $a \oplus b = a + b - a \odot b$, where $\odot$ denotes element-wise multiplication. Then, g can be defined as:

$$g(e, e_0) = e_0 + \sum_p \bigoplus_{0 \le i \le n} \sum_{0 \le j \le k} w_{i,j}^p h_{i,j}^p(e), \tag{3}$$

where n is the number of rule templates that define predicate p, k is the number of rules generated from the i-th rule template, $w_{i,j}^p$ is the trainable weight associated with the j-th rule generated from the i-th rule template for predicate p, and $h_{i,j}^p(e)$ represents a single deduction step using the j-th rule generated from the i-th rule template for predicate p.

For a given learning task, NLRL generates a set of candidate rules for each rule template, but only one rule can be selected for each rule template. Therefore, each weight $w_{i,j}^p$ is obtained by applying a softmax function to the underlying vector of weights θ_i^p:

$$w_{i,j}^p = \text{softmax}(\theta_i^p)[j]. \tag{4}$$

Given a rule c, each function $h_c : E \to E$ takes a valuation representing the truth value of the ground atoms and computes a valuation that represents the truth value of the atoms resulting from the application of the rule. For each function h_c, NLRL builds the following matrix X_c:

$$X_c[r, m] = \begin{cases} x_r[m] & \text{if } m < |x_r| \\ (0,0) & \text{otherwise} \end{cases} \tag{5}$$
$$x_r = \{(a, b) \mid \text{satisfies}_c(\gamma_a, \gamma_b) \wedge \text{head}_c(\gamma_a, \gamma_b) = \gamma_r\},$$

where x_r is a set of pairs of indexes, and each pair refers to two ground atoms that entail the ground atom of index r. Each matrix X_c is a $n \times w \times 2$ matrix, where w is the maximum number of ground atoms that entail each ground atom. It is worth noting that the unused index pair $(0,0)$ must reference to a pair of atoms. Therefore, NLRL introduces the falsum atom $\bot$ in G, and it maps $(0,0)$ to $(\bot, \bot)$. In order to perform the forward chaining steps during the training phase, NLRL computes two slices of X_c, namely $X_1, X_2 \in \mathbb{N}^{n \times w}$, as:

$$X_1 = X_c[\_, \_, 0] \qquad X_2 = X_c[\_, \_, 1]. \tag{6}$$

NLRL retrieves the actual truth values of each ground atom to obtain $Y_1, Y_2 \in [0,1]^{n \times w}$ using $\text{gather} : E \times \mathbb{N}^{n \times w} \to \mathbb{R}^{n \times w}$ as follows:

$$Y_1 = \text{gather}(e, X_1) \qquad Y_2 = \text{gather}(e, X_2). \tag{7}$$

Then, NLRL builds a new matrix $Z_c \in [0,1]^{n \times w}$ defined as:

$$Z_c = Y_1 \odot Y_2. \tag{8}$$

Finally, NLRL defines $h_c(e)$ as:

$$h_c(e) = e' \qquad \text{where} \qquad e'[k] = \max_{1 \le j \le w} Z_c[k, j] \tag{9}$$

Note that the maximum operator implements fuzzy disjunction while element-wise multiplication provides for fuzzy conjunction.

2.3. The proposed method

In order to generate a set of suitable candidate rules to solve a reinforcement learning task, NLRL requires the user to specify a large amount of data about the structure of the solution. On the contrary, SD-NLRL requires the complete set of possible states, and it automatically generates a set of abstract rules that can represent the solutions of the learning task.

The rule generation process that characterizes SD-NLRL is described as the `generate_rules` function in Algorithm 1. This function requires the set of possible action atoms A, the set of possible states S, and the set of background atoms B. The function subdivides each state into a set of groups V using the `get_groups` function. Each group contains a subset of the state, and each atom in a group is, directly or indirectly, connected with the other atoms in the same group through its constants. Therefore, different groups have different constant sets, and each group represents a feature of the state that contributes to the truth value of the action predicate. Then, for each pair of group and action, the `generate_rules` function retrieves the constants that are shared between the action atom and the group using the `get_shared` function. The `generate_rules` function inserts into the group the background atoms that include at least one shared constant. In fact, the shared constants are the most significant constants because they relate the features of the state with the action to be taken. Then, `generate_rules` function computes, using the `get_free` function, the set of free constants, which are the constants that are not shared between the action atom and the group after the inclusion of the related background atoms. If the set of free constants is empty, the `generate_rules` function computes the final rules using the `unground` function, which transforms each ground term of the rule into a corresponding variable. Otherwise, the function produces a rule for each free constant. In particular, the function transforms the free constants that are different from the one taken in consideration. Then, it includes in the body of the generated rule the background atoms that contain the considered free constant. Finally, the `unground` function transforms the remaining constants in the rule into variables. The generation of a new rule for each free constant is useful to reduce the complexity of each rule. In fact, the goal of the function is to extract the most simple features from each state and let the differentiable architecture select the most appropriate subset of the rules that is capable to solve the learning task. The `unground` function starts from the body of the rule given as second argument, and it proceeds from left to right. Then, it transforms the head of the rule given as first argument. The third argument of the `unground` function is a set of constants. If the set is not empty, the function transforms only the constants that are included in this set.

Algorithm 1 The function that performs the generation of rules in SD-NLRL

```
function generate_rules(A, S, B)
    rules ← {}
    for each s ∈ S do
        V ← get_groups(s)
        for each g ∈ V do
            for each a ∈ A do
                shared ← get_shared(a, g)
                insert atoms b ∈ B into g if constants(b) ∩ shared ≠ ∅
                free ← get_free(a, g)
                if free = ∅ then
                    r ← unground(a, g, {})
                    rules ← rules ∪ r
                else
                    for each f ∈ free do
                        fixed ← free \ {f}
                        partial_rule ← unground(a, g, fixed)
                        insert b ∈ B into the body of partial_rule if f ∈ constants(b)
                        r ← unground(a, body(partial_rule), {})
                        rules ← rules ∪ r
                    end for
                end if
            end for
        end for
    end for
    return rules
end function
```

The discussed function used by SD-NLRL for the generation of rules allows an action predicate to be defined by an unpredetermined number of rules. Moreover, the body of each rule can consist of an arbitrary number of atoms, and no limits on the arity of the predicates in these atoms is fixed. The differentiable architecture of NLRL is modified in SD-NLRL to coherently support the changes of the rule generation process. In particular, the deductive process is modified, and SD-NLRL defines a single deduction step g as:

$$g(e, e_o) = e_o + \sum_{p} \bigoplus_{0 \le j \le k} w_j^p h_j^p(e), \tag{10}$$

where w_j^p is the weight associated to the j-th rule defined for predicate p, and $h_j^p(e)$ represents a single deduction step that uses the j-th rule defined for predicate p. Note that, in order to ensure that the weights w_j^p are in $[0, 1]$, each weight w_j^p is normalized:

$$w_j^p = \frac{\max(w^p) - w_j^p}{\max(w^p) - \min(w^p)}. \tag{11}$$

Moreover, the definition of the deduction step, represented by $h_j^p(e)$, has been coherently modified in SD-NLRL. In particular, SD-NLRL defines a matrix called $X_c \in \mathbb{N}^{n \times w \times d}$ for each rule c, as follows:

$$X_c[r,m] = \begin{cases} x_r[m] & \text{if } m < |x_r| \\ (0,\dots,0) & \text{otherwise} \end{cases} \tag{12}$$
$$x_r = \{(a_1,\dots,a_d) \mid \text{satisfies}_c(\gamma_{a_1},\dots,\gamma_{a_d}) \wedge \text{head}_c(\gamma_{a_1},\dots,\gamma_{a_d}) = \gamma_r\}.$$

SD-NLRL builds the matrices $X_1,\dots,X_d \in \mathbb{N}^{n \times w}$ before the training phase, where X_i represents the i-th element of the body of the rule c. Then, SD-NLRL obtains the matrices $Y_1,\dots,Y_d \in \mathbb{R}^{n \times w}$ using the gather function. These matrices are used to build the matrix Z_c as:

$$Z_c = Y_1 \odot \cdots \odot Y_d. \tag{13}$$

Finally, SD-NLRL defines $h_c(e)$ in the same way as NLRL does.

The weight normalization rule that SD-NLRL uses allows an action predicate to be defined by multiple rules. In particular, each weight depends only on the minimum weight and on the maximum weight defined for the predicate. Therefore, SD-NLRL can associate the same weight with different rules. However, the adopted normalization technique does not work well when SD-NLRL generates only one rule for an action predicate. Moreover, this normalization technique implies that a rule with weight 0 and a rule with weight 1 are always learned, which can prevent SD-NLRL from learning the optimal strategy in some cases.

3. Experimental results

In order to assess the performance of SD-NLRL, the proposed method has been used on the reinforcement learning tasks that are discussed in [7]. In particular, the considered tasks are: CLIFFWALKING, WINDYCLIFFWALKING, UNSTACK, STACK, and ON. In the CLIFFWALKING task, the agent must go from a start position to a goal position of a 5×5 grid without reaching a cliff. The WINDYCLIFFWALKING task is similar to CLIFFWALKING, but the agent has a 10% chance of going downwards, no matter which action it takes. The other three tasks require the agent to manipulate blocks. In particular, STACK requires the agent to pile up all the blocks in a single column, UNSTACK requires the agent to move all blocks to the floor, and ON requires the agent to move a specified block onto another specified block. For a fair comparison between NLRL and SD-NLRL, SD-NLRL was tested using the task configurations documented in [7].

Note that, as previously discussed, SD-NLRL tries to subdivide each state into disjoint groups of atoms, but the states of block manipulation tasks are composed of densely connected atoms. Therefore, SD-NLRL generates a large number of complex rules. Unfortunately, the required computational resources increase considerably as the size of the rules and the number of rules increase. Therefore, SD-NLRL fails to complete the training phases of the block manipulation tasks, and only the cliff-walking tasks are discussed in the remaining of this section.

The implementation of SD-NLRL is based on the official implementation of NLRL[1]. In particular, the implementation of SD-NLRL and the implementation of NLRL share the same

[1] github.com/ZhengyaoJiang/NLRL

Table 1
A performance comparison between NLRL and SD-NLRL on the CLIFFWALKING and on the WINDY-CLIFFWALKING tasks. The optimal values are taken from [7].

Environment	Task	NLRL	SD-NLRL	Optimal
CLIFFWALKING	training	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
	center	0.859 ± 0.050	0.775 ± 0.198	0.920
	6 by 6	0.841 ± 0.024	0.631 ± 0.381	0.860
	7 by 7	0.824 ± 0.024	0.520 ± 0.514	0.840
WINDYCLIFFWALKING	training	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
	center	0.672 ± 0.579	-0.350 ± 0.430	0.868 ± 0.303
	6 by 6	0.345 ± 0.736	-0.991 ± 0.093	0.748 ± 0.135
	7 by 7	0.506 ± 0.528	-1.012 ± 0.047	0.716 ± 0.181

hyper-parameters. The only exceptions are the learning rate and the number of deduction steps. The learning rate of SD-NLRL was set to 0.0005 and 0.0001 for CLIFFWALKING and WINDYCLIFFWALKING, respectively. The number of deduction steps of SD-NLRL was set to 1 for both tasks because a single forward chaining step is sufficient to obtain the correct truth values of the action predicates.

Table 1 shows the performance of SD-NLRL for the two considered cliff-walking tasks. The results shown in Table 1 were obtained performing 5 runs for each task, and each trained model was evaluated on 100 episodes for each task. Then, the results were averaged over all runs. In order to evaluate the generalization capabilities of SD-NLRL, the trained models were tested on other tasks that are slightly different from the ones used for training. In particular, the algorithm was evaluated on the same tasks discussed in [7]. The considered variants of cliff-walking tasks are five: *top left*, *top right*, and *center*, which change the starting position of the agent, and 6 *by* 6 and 7 *by* 7, which use a larger grid size.

The results reported in Table 1 show that SD-NLRL is able to learn an effective strategy that solves CLIFFWALKING. Moreover, the learned model is able to generalize to different tasks. However, the results suffers from great variance. In fact, in 1 of 5 five trials, SD-NLRL learned a strategy that is only able to occasionally get a positive reward. In particular, when the agent does not succeed to quickly learn a good strategy, it remains trapped in a local optimum and stops learning. The results for WINDYCLIFFWALKING show even worse performance, and SD-NLRL consistently fails to learn a good strategy. The algorithm learns over time, but the learning speed is very slow, and it always remains trapped in a local optimum. This behavior is explained by the fact that in NLRL rules are generated from a carefully hand-crafted program template. On the contrary, SD-NLRL automatically generates the rules from the states of the environment, trying to limit both the size and the number of generated rules. Therefore, SD-NLRL does not succeed in generating the best rules for each action predicate. Both these problems represent interesting challenges for the future.

In order to offer a comprehensive view on SD-NLRL, the learned rules that obtain the best performance in the CLIFFWALKING task (on the right) with their corresponding weights (on the left) are reported:

```
1.0     down() :- current(Y,X), last(Y), succ(Z,Y).

1.0     left() :- current(X,Y), succ(Y,Z), zero(Y).

1.0     right() :- current(Y,X), succ(Z,Y), succ(Y,M).
0.99    right() :- current(X,Y), succ(Z,Y), succ(Y,M).
0.98    right() :- current(X,Y), last(Y), succ(Z,Y).
0.70    right() :- current(X,X), succ(Y,X), succ(X,Z).

1.0     up() :- current(X,X), succ(X,Y), zero(X).
0.44    up() :- current(X,X), last(X), succ(Y,X).
```

The learned rules represent an effective strategy, and the agent obtains high returns on the training environment (0.844 ± 0.034). However, the learned strategy is not compact, and some rules are unnecessary. For example, the last two rule defined for `right()` and the last rule defined for `up()` can be safely omitted. SD-NLRL generates the rules directly from the states of the environment, and many generated rules are unnecessary or even harmful and they should be avoided. Therefore, an interesting research direction for the future regards the reduction of unnecessary rules either refining the rule generation process or introducing a penalization for redundant rules. It is worth noting that the only rule learned for `left()` can compromise the performance of the strategy because the agent often moves left before moving up. Therefore, the learning of strategies when actions are forbidden should be further investigated. Finally, the learned rules, together with their weights, that obtain the worst performance in the CLIFFWALKING task are reported for completeness:

```
0.58    down() :- current(X,X), last(X), succ(Y,X).
1.0     down() :- current(Y,X), last(Y), succ(Z,Y).
0.4     down() :- current(X,X), succ(Y,X), succ(X,Z).
0.67    down() :- current(X,Y), succ(Y,Z), zero(Y).

0.51    left() :- current(X,X), succ(X,Y), zero(X).
1.0     left() :- current(Y,X), succ(Z,Y), succ(Y,M).

1.0     right() :- current(Y,X), succ(Z,Y), succ(Y,M).
0.82    right() :- current(X,X), last(X), succ(Y,X).
1.0     right() :- current(X,Y), last(Y), succ(Z,Y).
0.42    right() :- current(X,X), succ(Y,X), succ(X,Z).

1.0     up() :- current(X,Y), succ(Y,Z), zero(Y).
0.49    up() :- current(Y,X), succ(Y,Z), zero(Y).
1.0     up() :- current(X,X), succ(Y,X), succ(X,Z).
```

In this case, the number of learned rules is higher, and the agent obtains low returns in the training environment (0.154 ± 0.629). In fact, many rules are unnecessary or even wrong. Note that, in both cases, only the rules with a weight greater than 0.3 are shown for space limitations.

From the point of view of the required computational resources, SD-NLRL drastically reduces the number of generated rules with respect to NLRL. Moreover, SD-NLRL performs only 1 deduction step, thus further reducing the required computational resources. In fact, NLRL generates 2813 rules for the cliff-walking tasks, while SD-NLRL generates only 36 rules for the same tasks. The number of generated rules directly influences the required amount of computational resources. Actually, NLRL requires approximately 10 days to complete a training process, while SD-NLRL complete a training process in less than 2 hours. The required memory is reduced as well. SD-NLRL uses approximately 1.6Gb, while NLRL uses approximately 17Gb. These results are preliminary, and a complete analysis of the computational resources that SD-NLRL requires is left for the future. However, it is worth noting that SD-NLRL does not always reduce the required computational resources because it struggle with densely connected states. In fact, in many cases, SD-NLRL generates a large amount of complex rules because it does not limit the size and the number of generated rules. Therefore, the generation of compact and abstract rules even in worst cases represents an important direction for future research.

4. Conclusion

This paper introduced a novel neural-symbolic method for reinforcement learning called SD-NLRL. The proposed method is based on NLRL, but it generates candidate rules directly from the states of the environment. The experimental results discussed in the final part of this paper show that the proposed method is able to effectively learn a good strategy in one of the cliff-walking tasks, and that the method is also able to generalize this strategy to tasks that are slightly different from the one used for training. However, SD-NLRL fails to solve block manipulation tasks because it is not able to generate compact rules from the densely connected states that characterize these tasks. The generation of compact, yet general, rules from densely connected states represents an important challenge for the future. Moreover, SD-NLRL often remains trapped in local optima, and this is unquestionably evident when the difficulty of the task under investigation increases. Another important challenge for the future is to make SD-NLRL more robust against the traps represented by local optima. In addition, the analysis of the learned rules suggests that: SD-NLRL learns many unnecessary rules, and it is not able to learn an effective strategy when an action is forbidden. Both these problems are two other relevant challenges for the future. Finally, the assumption that the environment is able to provide the set of all possible states represents another important limitation of SD-NLRL. Therefore, a relevant improvement of the method regards the design of an iterative rule-generation process in which new rules are generated only when new states are encountered. In summary, this is a preliminary work, and the current form of the proposed method cannot be considered competitive with other neural-symbolic methods for reinforcement learning like NLRL. However, the presented experimental results are encouraging, and an improved version of SD-NLRL can be considered as a viable means toward effective neural-symbolic reinforcement learning.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with deep reinforcement learning, Preprint arXiv:1312.5602 (2013).

[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] 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.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., Mastering the game of Go without human knowledge, Nature 550 (2017) 354–359.

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

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

[7] 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.

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

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

[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] 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.

[13] 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.

[14] 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.

[15] A. Payani, F. Fekri, Inductive logic programming via differentiable deep neural logic networks, Preprint arXiv:1906.03523 (2019).

[16] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, D. Zhou, Neural logic machines, Preprint arXiv:1904.11694 (2019).