Domain Informed Oracle for Reinforcement Learning

SAMAR RAHMOUNI, Carnegie Mellon University, Qatar
GISELLE REIS, Carnegie Mellon University, Qatar

Reinforcement learning (RL) is a powerful AI technique that does not require pre-gathered data but relies on a trial-and-error process for the agent to learn. This is made possible through a reward function that associates state configurations to a numerical value. The agent's goal is to maximize its cumulative reward over its lifetime.

Unfortunately, there is no systematic method to design a reward function since interpreting abstract states in RL in the context of a domain needs to be done on a case by case basis.

This is crucial because an informed reward function that rewards different configurations, on top of terminal states, is shown to result in efficient training.

We propose a *Domain Informed Oracle (DIO)* to systematically incorporate domain specific knowledge into RL reward functions. DIO is a collection of domain specific rules written in a declarative language, such as Prolog. It does not rely on the RL representation of states, allowing the programmer to focus on the domain knowledge using an expressive and intuitive language, where states and rules can be defined conveniently. For each state and action pair, DIO provides information to the reward function, to dynamically adapt itself.

Our implementation is tested on a grid world with dynamic obstacles. and compared to a basic RL algorithm. Our results show that in our simple scenario, some weights to DIO ensure slightly better safety during training, while minimizing policy loss during deployement. Although the improvements are minimal, they are promising as the architecture we present allows for a seamless translation between a declarative programming language and a numerical reward function.

Our results need further investigation in more complex settings while allowing for the policies to converge before comparison.

CCS Concepts: • **Theory of computation** → *Constraint and logic programming*.

Additional Key Words and Phrases: probabilistic logic programming, reinforcement learning, reward shaping

ACM Reference Format:
Samar Rahmouni and Giselle Reis. 2022. Domain Informed Oracle for Reinforcement Learning. 1, 1 (November 2022), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Implementing a robust adaptive controller that is effective in terms of precision, time, and quality of decision in dynamic and uncertain scenarios has always been a central challenge in AI and robotics. When autonomous agents are deployed in the real world, we want to ensure that they are able to adapt to unforeseen scenarios, as well as keep their efficiency. This efficiency is measured in terms of optimality of actions and time to make a decision. Since we are unable to provide a repertoire of all possible scenarios and actions, agents need to be able to autonomously predict and adapt to changes. Reinforcement Learning (RL) is an approach that supports dynamically adapting

Authors' addresses: Samar Rahmouni, srahmoun@andrew.cmu.edu, Carnegie Mellon University, Doha, Qatar; Giselle Reis, giselle@cmu.edu, Carnegie Mellon University, Doha, Qatar.

XXXX-XXXX/2022/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to new input [21]. It has beed successfully used by AlphaGo, Deepmind AlphaStar, and OpenAI Five to solve Go, StarCraft II and Dota 2 [16]. Reinforcement Learning is a powerful tool as it does not require pre-gathered data as most Machine Learning (ML) techniques do. The general idea of RL is learning via trial-and-error, guided by a *domain dependent* reward function. For example, if the agent is a self-driving car, the reward function would greatly penalize states where it crashes. However, this means that the car is bound to crash to learn not to crash again. A better reward function can include the physics equations to predict, with some degree of certainty, the car's trajectory for the next few seconds. By looking into the future, the reward function can penalize bad behavior before it reaches a catastrophic state (a crash). A better reward function prunes the (often infinite) search space faster, allowing the agent to explore (breadth) new states instead of exploiting (depth) dead ends. The task of choosing the reward function is thus crucial, yet difficult as shown in 2.1. In this work, we propose a Domain Informed Oracle (DIO) written in a declarative language to inform a reinforcement learning algorithm. Our method provides a systematic way to encode domain specific rules into a reward function for RL that does not rely on the state representation within the RL algorithm.

2 MOTIVATION

Reinforcement Learning is a method of learning that maps situations to actions in order to maximize its rewards [21]. Rewards are numerical values associated to a state and action. Precisely, one defines a reward function $R : (S \times A) \rightarrow \mathbb{R}$ where S defines the state space and A the action space. This function is equivalent to $R' : S \rightarrow mathbbR$ where the queried state is the one resulting from a given (state, action) pair. Those can be used interchangeably. More precisely, in the case we describe, the state refers to the current configuration of the environment and the action refers to the action chosen by the RL agent. By defining this reward function and the scenario of the problem the agent is trying to solve, reinforcement learning has the advantage of not requiring a prior dataset. Indeed, the agent is not told what to do, but rather learns from the effect of its actions on the environment.

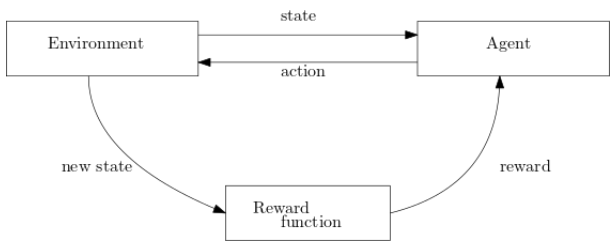

Fig. 1. Reinforcement Learning Routine

The diagram in Figure 1 is a high-level description of how an agent using reinforcement learning can be trained. The upper left box represents the *environment* as seen by the agent according to its sensors. The current state of the environment is represented as a *state vector*. At each iteration, the agent will receive the state vector as input, and needs to choose an *action* to take. Once the action is taken, the environment is updated to the next state and the agent receives a *reward* as feedback. This reward is a domain dependent function that represents how "good" the new state is. The agent's goal is to increase its reward by taking actions that reach better states each time. The triple (state, action, reward) helps the agent in shaping the final policy.

The reward function is a crucial aspect of the RL algorithm. For instance, consider a game of chess where the agent is punished when it loses and rewarded if it wins. The agent is bound to learn

how to maximize its winnings but it will need to exhaust multiple possible combinations to learn. In this case, the training time is not optimal. A better approach would be to also reward it for making a good opening, for instance. Another example would be only considering negative rewards. Say we want our agent to escape a maze, and we punish it at every timestep for not escaping. If there is a fatality state (*e.g.*, a fire or a black whole), the agent will learn to move towards the fatality state as to cut its negative rewards as soon as possible. In conclusion, a good reward function is the first step of optimal learning. By choosing a *refined* reward function, we can ensure a faster and more efficient training [14], possibly with fewer errors.

2.1 Reward Shaping

Reward shaping [15] refers to the lack of systematic methods to design a reward function that ensures fast and efficient learning [14]. This generation of an appropriate reward function for a given problem is still an open challenge [13]. The importance of a reward function for efficient training is shown in [14]. A sparse reward function defined as a *goal-reward representation* is one where the agent is only rewarded for entering a goal state, while a dense reward function is defined as an *action-penalty representation*, precisely, one where the agent is penalized for every action it executes. [14] shows that a denser reward structure improves performance. Moreover, an informed reward function is able to sufficiently deter the exploration of undesirable states while encourage the exploitation of desirable ones, continuously adapting to acquire knowledge and resolving the conflict when necessary. Precisely, an informed reward function can tackle the *exploration vs. exploitation dilemma*, a central challenge in RL [11]. In particular, an agent that can continuously acquire knowledge is responding to the uncertainties of the world, where states can never be exhausted. This involves an adaptive learned policy that responds to conditions and tasks that were not encountered in the past [9, 20].

Ideally, rewards would be given by the real-world, i.e. *native rewards*. For instance, recent work investigates dynamically generating a reward using a user verbal feedback to the autonomous agent [22]. However, most RL agents can only stay in simulation because the trial-and-error nature of RL prevents any guarantees of safety. Thus, there exists a need for *shaping rewards* instead. Unfortunately, due to the lack of systematic methods to build a denser reward function, it is common to use a sparse reward function as long as it still guarantees convergence.

3 RELATED WORK

To tackle the reward shaping challenge from Section 2.1, we are inspired by the current Neurosymbolic AI trends, which explore combinations of deep learning (DL) and symbolic reasoning. The work has been a response to criticism on DL's lack of formal semantics and intuitive explanation, and the lack of expert knowledge towards guiding machine learning models.

Current Neurosymbolic AI trends are concerned with knowledge representation and reasoning, namely, they investigate computational-logic systems and representation to precede learning in order to provide some form of incremental update, e.g. a meta-network to group two sub-neural networks [2]. As a result, neurosymbolic AI has been successfully applied to vision-based tasks such as semantic labeling [12, 23], vision analogy-making [17], or learning communication protocols [7]. In general, neurosymbolic AI trends show promising results in improving ML algorithms, whether that is from an interpretability aspect or an optimization one. More recent works take this trend and incorporate symbolic reasoning and domain knowledge in reinforcement learning settings [1, 5, 8, 18]. [8, 18] use the general idea of *reward shaping* and *epsilon adaptation* respectively to incorporate procedural knowledge into a RL algorithm. Both works introduce this combination as a successful strategy to guide the exploration and exploitation tradeoff in RL. They both show promising results. While [8] focuses on providing formal specifications for reward shaping, it

lacks practical consequences to the implementation of most RL to make use of its formal methods conclusions. On the other hand, [18] proposes a method to adapt ϵ based on domain knowledge, the method is specifically applied to "Welding Sequence Optimization". To do so, the RL algorithm is modified in itself, similarly to what was done in [5]. Precisely, in [5], the RL algorithm itself is modified to deal with states that are model-based as opposed to vectors. They defined their method as Relational RL. Furthermore, they conclude that by using more expressive representation language for the RL scenario, their method can potentially offer a solution to the problem of meta-learning. While [5, 18] both present promising rewards, they lack the modularity necessary for scaling the proposed methods to further RL implementations.

This is further reinforced by more recent work, precisely, *reward machines* that define an automaton to adapt a reward function given step transitions [10]. By exposing the structure in the reward function, [10] shows that this enables to find solutions faster. However, given the nature of a state machine, reward machines are unable to adapt to the uncertainties of the world.

To face those limitations, DIO does not rely on an abstract representation to infer a reward function, rather only needs to care about the translation to a domain specific language, like prolog or datalog to assess a given world. The stochasticity of the world is then inherent, given a probabilistic logic program. DIO provides a more declarative approach to reason about rewards, thus providing a systematic method to map labels to rewards.

4 DOMAIN INFORMED ORACLE

We tackle the challenge of finding a systematic method to map a state to a numerical value: a reward. To do so, we make use of the fact that rewards are domain dependent and thus, given domain specific rules, a *domain informed* module can guide a RL agent towards better decisions. This can be done by adapting the reward function. For instance, we consider defining which states are desirable, which are to be avoided and which are fatal. Given rules and judgments, a logic programming module is able to search the space and send feedback to the reinforcement learning agent. The goal is a systematic method to design a reward function which can ensure faster and more efficient training. This knowledge can furthermore be incorporated into resolving the exploration vs. exploitation dilemma. For instance, if a domain informed module can infer that only one of the possible next states is desirable, then exploration in that specific case is suboptimal. We will call the proposed module a *Domain Informed Oracle* (DIO).

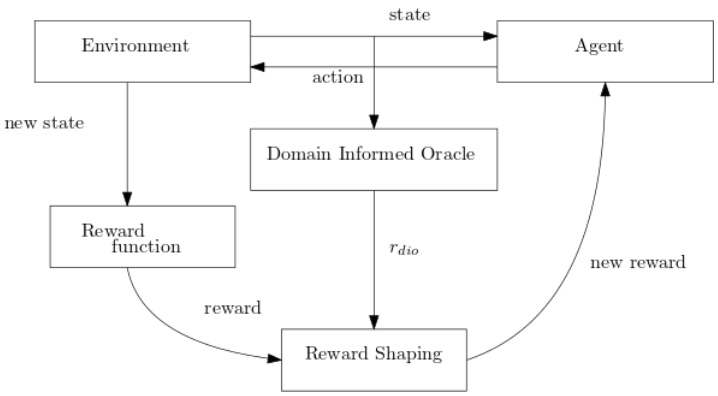

Fig. 2. Overview of the proposed solution

The diagram in Figure 2 is a high-level description of our proposed solution. Precisely, the (state, action) pair is fed into DIO. While the environment produces its own reward function, DIO also computes its own reward.

We could leave the *reward shaping* unit as a design choice. However, it will be argued in 4.4 that whether it is left as a design choice or else, is irrelevant given how DIO affects the final results.

4.1 Architecture

In this section, we lay the foundations of the architecture that combines the Domain Informed Oracle with reinforcement learning. Note that in our proposed architecture, we suppose Proximal Policy Optimization [19]. It does not mean that our solution is specific to it, rather it can be generalized to any algorithm choice.

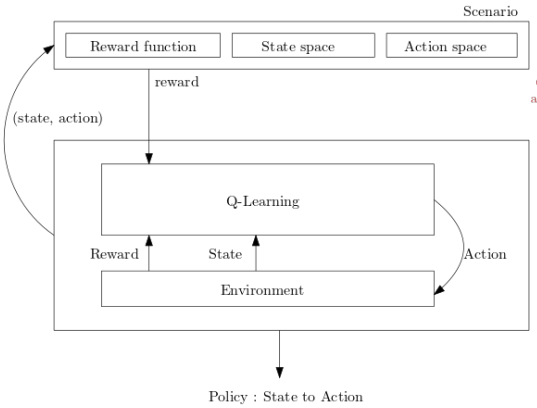

Fig. 3. Reinforcement learning architecture

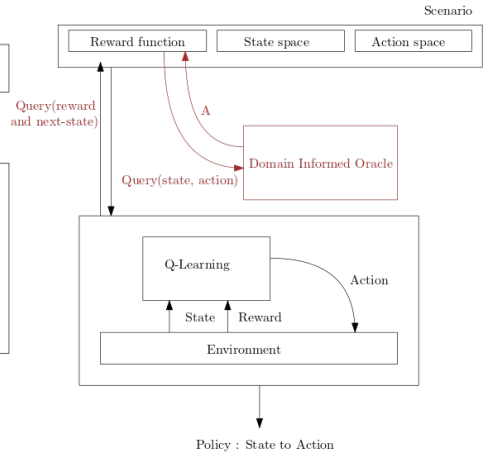

Fig. 4. DIO+RL architecture

The diagram in Figure 3 describes the basic routine of RL in more detail. The environment defined by the scenario sends the current state to the *Proximal Policy Optimization* algorithm. The agent chooses an action from the action space and sends it to the environment. This action affects the environment stepping it to some next state. The resulting state along its associated reward is computed from the reward function and step function formalized in the scenario. Thus, in the next iteration, the agent receives the reward from its previous action which it uses to improve its policy and continues with its training starting from the computed next state.

The architecture in Figure 4 introduces DIO in the feedback loop. It is kept independent of the RL module. Precisely, when the scenario is query-ed for the reward and the resulting next state of a (state, action) pair, rather than computing the reward using the reward function, the latter is able to query DIO. The result of this query is A: a numerical value we defined as r_{DIO}, the reward given by DIO to the (state, action) pair. More on how this reward is computed in 4.4.

4.2 DIO procedure

In practice, we consider the following modules and their interactions as shown in 5.

(1) **World Rules** defining the rules governing the world. This is domain-dependent and implemented in a logic programming file, i.e. we are able to define the next step via step semantics.
(2) **Knowledge Base** defining the ground facts which describe the world at a given time step. This module is continuously updated to account for the dynamics of the state.
(3) **Labels** i.e., textual "norms" corresponding to an iteration of the state. In practice, they are all possible judgments on the resulting state, e.g. *crash :- obs(X,Y), agent(X,Y),* or *maybecrash :- nextObs(X,Y), agent(X,Y).* Those labels have probabilities associated with them and indicators. Precisely, an indicator over a negative state is -1.
(4) **Translation Unit** defining the translation from state to ground facts and from labels to a numerical value, e.g. if the predicate crash is true with $P = 0.25$, then the reward shaped is $r + (-0.25)$.
(5) **Reinforcement Learning** is our independent module that does not make assumption on the algorithm chosen for RL.

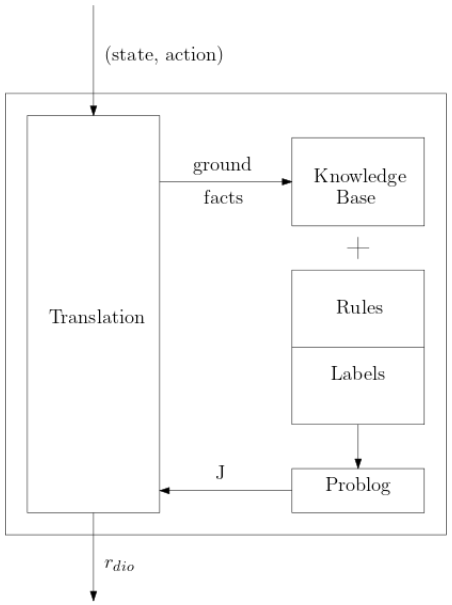

Fig. 5. Modules & Interactions

Figure 5 describes the interactions of the different modules. Precisely, given the rules and the world knowledge base at a given time t, we are able to produce the corresponding label, i.e. the query over a predicate. The predicate is fed into the Translation Unit (TU) that transforms the predicate to a numerical value that is given to the Reinforcement Learning as a reward shaping r'. Note that the inference on the query is done by a declarative tool that incorporates probabilities called *Problog* that we introduce in 4.3.

4.3 Problog Procedure

Problog is a logic programming language that aims to bridge between probabilistic logic programming and statistical relational learning [6].

Definition 4.1 (Statistical Relational Learning (SRL))**.** Discipline of Artificial Intelligence that considers first order logic relations between structures of a complex system and model it through probabilistic graphs such as Bayesian or Markov networks.

Definition 4.2. Probabilistic Logic Programming (PLP) Discipline of Programming Languages that augments traditional logic programming such as Prolog with the power to infer over probabilistic facts to support the modeling of structured probability distributions.

A problog program specifies a probability distribution over possible worlds. This probability distribution corresponds to the possible worlds whether a fact is taken or discarded given the probability associated with it. Precisely, they define a world as a subset of ground probabilistic facts where the probability of the subset is the product of the probabilities of the facts it contains. Furthermore, problog extends PLP with the power of considering evidences in the inference task. This is made possible without requiring the transformation the Bayesian networks on which to use SRL. Instead, problog considers the subset described above and assumes only worlds where the evidence held remains true. Those possible worlds and their associated probabilities are then added and divided by the choice with the higher probability. Problog makes this possible by a 3-steps conversion from a problog program to a weighted boolean formula.

First, problog grounds the program by only considering facts relevant to the query in question. The relevant ground rules are specifically converted to equivalent Boolean formulas. Precisely, inferences are converted into bi-directional implications and its corresponding premises are converted to a conjunction of disjunction of facts. Finally, problog asserts the evidence by adding it to the previous boolean formula as a conjunction and defines a weight function that assigns a weight to every literal. The weights are derived from the probability associated with the relevant literal, whether explicility given or implicility computed.

4.4 Computation of Final Reward

In general, reward shaping is expressed as follows $R = r_{rl} + r'$, such that r_{rl} is the original reward defined by the reinforcement learning environment. Usually, this associates the value of 1 for a positive terminal state, and -1 for a negative terminal state. r' is the human bias, i.e. what is added to "shape" the reward function. Although only rewarding terminal states is enough to reach a policy that will minimize its losses, it is important to note that a reward function should consider the cost of a step for optimal and fast training. The cost of a step is dependent on the state given at that step. For instance, an autonomous vehicle stopping with no cars around is different from a vehicle stopping while a car is behind it. Although the states are not *terminal*, i.e. they do not end an episode, they have different weights. An optimal reward function is one that considers those differences. A logic programming module allows us to infer over states using a knowledge base and world rules seamlessly.

Let's consider the following example in Figure 6. Our agent is running away from a predator. It is currently at position $(0, 0)$, and decides to go right, to escape. Since, it is a stochastic environment, there is a 0.1 chance that his parts might fail and he stays in the same spot, thus getting eaten by the predator. This is easily expressed in *problog* as follows.

```
0.9 :: pos(X+1, Y) :- direction(right), pos(X, Y).
0.1 :: pos(X, Y) :- direction(right), pos(X, Y).
```

Consequently, there is a 0.9 chance that we end up in a 'desirable' state, and a 0.1 chance that we end up in an 'undesirable' state. Therefore,

$$r'_{\text{DIO}} = \sum_{L} Pr[L \mid (s_t, a_t)] I[L]$$

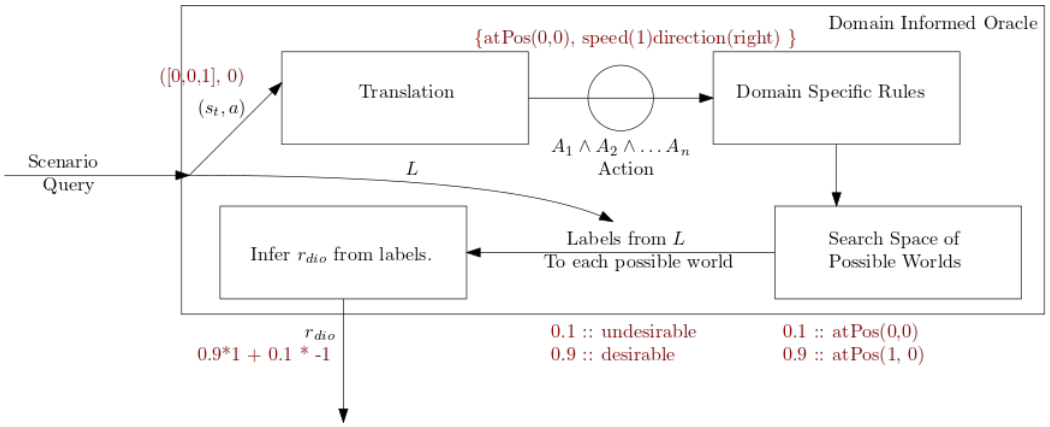

Fig. 6. Step-by-step of Computed Final Reward

where desirable and undesirables are labels noted by L. We identify their indicator $I[L]$ by 1 if positive, -1 if negative. This is equivalent to the expectation over a (state, action) pair computed by the logic programming module.

It is the case that this reward has to be normalized to the weight of a step. Indeed, suppose there are S possible states for the agent to be at. The accumulated reward of the steps should at most equal that of the terminal states. Thus, the final reward,

$$r_{\text{DIO}} = \frac{r'_{\text{DIO}}}{S} \qquad \text{and} \qquad R = r_{rl} + \alpha r_{\text{DIO}}$$

S is the size of the observation space. In particular, since the reward is normalized as a cost of a step, thus, always allowing terminal states to have more value.

Note that the final reward from DIO is added considering some α. This is precisely for comparison purposes, i.e. how much value should we give the logic programming module to inform a step.

5 DYNAMIC OBSTACLES IN A GRIDWORLD : WORKING EXAMPLE

For our experiments, we take the following scenario. An agent exists in a 100x100 grid. There are n dynamic obstacles around that have uniform chances of choosing any direction to move in. The agent can make a decision to move either right, left, top or bottom, as well as not moving at all. There is a goal in the bottom-right of the grid. The agent has two goals: (1) survive by not crashing with the obstacles and (2) reach the goal in the lowest number of steps possible.

5.1 Scenario in Reinforcement Learning

This environment offered by gym-gridworld [3] is useful for testing our algorithm in a Dynamic Obstacle avoidance for a partially observable environment. Precisely, we define the state as follows.

$$S_t = [x, y, d, G]$$

(x, y) define the position of our agent while d its direction. G is the gridworld observed by the agent which includes walls, obstacles and free squares. Note that this observed environment is from the point of the view of the agent and does not represent the entire grid. The action space is,

$$A_t = \{\text{right} : 0, \text{up} : 1, \text{down} : 2, \text{left} : 3\}$$

Finally, the reward is a straightforward one that reward 1 for reaching the goal, -1 for failing to do so, either by colliding with an obstacle, or exhausting its battery (maximum number of steps).

Furthermore, for every step the agent takes, it has to spend the *cost of living*, which is $-1/n$, where n is the size of board.

5.2 Domain Specific Rules

The rules are defined in ProbLog [4]: a probabilistic prolog that allows us to capture the stochasticity of the environment, as we previously introduced in Section 4.3. Precisely, we want to consider the erratic movements of the obstacles, considering we do not have previous knowledge on the distribution of their given movement. We assume a uniform distribution and define the following. The rules of DIO deriving a predicate γ take the following form:

$$\frac{P_1 :: \psi_1 \qquad ... \qquad P_n :: \psi_n}{Q :: \gamma}\ \text{(action)}$$

where Q and $P_1, ..., P_n$ are probabilities for the facts γ and $\psi_1, ..., \psi_n$, respectively. The rule should be read as an implication from the top down, i.e., if the premise facts $\psi_1, ..., \psi_n$ hold with their corresponding probabilities, than we can infer the conclusion γ with probability Q. Note that Q will naturally be a function of $P_1, ..., P_n$. If there are k rules with conclusions $Q_1 :: \gamma, ..., Q_k :: \gamma$, then it must be the case that $\sum_{i=0}^{k} Q_i = 1$. The action is equivalent to our step semantics. In the gridworld example, we give the following.

$$(1)\ \frac{\text{atPos(X,Y)} \qquad \text{speed(V)} \qquad \text{timestep(T)}}{\text{atPos(X + V*T, Y)}}\ \text{(right)} \qquad (2)\ \frac{\text{obs(X,Y,V)} \qquad \text{timestep(T)}}{\text{0.2 :: obs(X + V*T, Y, V)}}\ \text{(time)}$$

(1) considers the movement of the agent while (2) considers the movement of the obstacles. Note that (2) considers a uniform distribution over the movement of the obstacle, since every obstacle has a uniform probability of moving up/down/left/right or staying in place. We could do the same for (1) by considering the probability of an action failing. In our case, we assume the movement is deterministic and no failure over the movement of the agent happens.

5.3 World Knowledge

Our world knowledge base covers the agent, the obstacles and the timestep. We consider two cases: *constant* ground facts vs. *dynamic* ground facts. The latter represents positions which are dynamically generated at every timestep while the former considers only the facts that remain true in every world, thus include the timestep, since we always move by 1-unit, and the speed, since the agent and the obtacles are defined to only move by 1-box every time. Given that our knowledge base Kb is defined by,

$$C = \{\text{speed}(1), \text{timestep}(1)\} \qquad D = \{\text{atPos}(X, Y), \text{obs}(X, Y, 1)\} \qquad Kb = C \cup D$$

5.4 From Norms to Labels

In the following, we define **norms** as textual sentences to describe the intended goal behavior. Thus, in the gridworld example, we consider one possible norm, (1) *crash* when the agent and the obstacle *possibly* overlap. That is if there is a chance of the obstacle and the agent choosing to move to the same square.

$$\frac{P_1 :: \text{atPos}(X, Y) \quad P_2 :: \text{obs}(X, Y, \_) \quad \dots}{P_1 \times P_2 :: \text{crash}}$$

Note that the final P associated to our label is exactly $Pr[L \mid (s_t, a_t)]$.

5.5 Translation Unit

The translation unit is the bridge between DIO and RL. It handles both feeding the world facts to DIO and translates the feedback given to a numerical value, through a given function. First, the DIO/RL loop is given in Algorithm 1.

Algorithm 1 DIO/RL Loop

```
1:  procedure STEP(S_t, a)                          ▷ (State, action) Pair
2:      Check for invalid actions
3:      Check for obstacles
4:      Update Obstacles positions
5:      Update Agent's position
6:      TU.UpdateWorld(position, direction, obstacles)   ▷ KB update in DIO
7:      obs, r_rl = Step'(S_t, ac)                  ▷ Call to Initial Env
8:      r'_DIO = getFeedback()                      ▷ Query DIO
9:      R = TU.getReward(r_rl, r'_DIO)
10:     return (obs, R)
```

Note that the translation unit first updates the knowledge base from DIO side. Given this update, it is possible to query DIO over the labels and their associated probabilities. Precisely, the *getFeedback* procedure computes r'_{DIO} the expected value over the possible worlds.

Algorithm 2 Inference of Judgment r'_{DIO}

```
1:  procedure GETFEEDBACK
2:      labels ← getLabels()                        ▷ Labels - Associated Probabilities
3:      P ← {crash}                                 ▷ Set of possible worlds
4:      I ← {−1}                                    ▷ Indicators associated with world
5:      i ← 1
6:      r ← 0
7:      while i ≤ length(labels) do
8:          r ← r+ labels[P[i]]*F[i]                ▷ Equivalent to the Expectation
9:          i ← i + 1
```

Finally, the translation unit **TU** can compute the final reward r'_{DIO} by normalizing it given the number of states and multiply it by a chosen α, before adding it to the original reward r_{rl}.

6 METHODOLOGY

6.1 Metrics

We compare our architecture incorporating DIO to a reinforcement learning architecture. Precisely for our scenario, we judge *optimality*, i.e. the number of steps to reach the goal and *safety*, i.e. ratio of successes to failures. Those metrics are analyzed during both training and deployment. We gather the cumulative rewards and cumulative rewards as well as the total number of failures and success over 180k frames of training, and 100 episodes of deployment.

6.2 Settings

Our above metrics and gathered for 8 different settings. We consider both α, 0 (equivalent to rl alone), 0.5, 1, 3 and 9. Similarly, we consider two settings for the number of obstacles: (1) minimal

(5% of the board is covered by obstacles), and (2) intermediate (10%). For each, we gather the metrics described above.

6.3 Gathered Results

During Training. For optimization purposes, training is done using multiprocessing reinforcement learning as indicated in figure 7. The network is loaded and parallel environments are spawned given the available number of processes. For n frames, the rl routine is returned before the final experience is collected and used to update the parameters of the neural network. For our purpose, the cumulative reward is the reward of each (obs_i, r_i) added over the number of updates. Similarly, the terminal states are computed according to this similar philosophy, meaning that at every given environment, if the resulting state is terminal and a failure, we add it to the cumulative failures to assess safety overtime. At the end of training, we compute the ratio of successes over failures to analyze the needed number of failures before convergence.

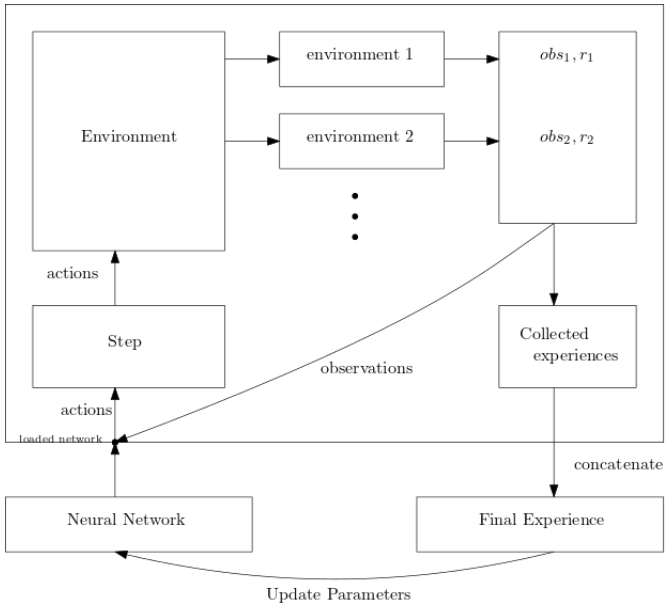

Fig. 7. Multi-processing Reinforcement Learning

During Deployment. Once training converges and a resulting policy is computed, we deploy the given policy for a 1000 episodes, given an episode ends either with a success or a failure. We compute the average number of steps it takes to reach the goal and its standard deviation to assess the optimality of the resulting paths. For safety, we still consider the ratio of success over failures as an indication.

7 RESULTS

The results will proceed by an analysis of the metrics discussed previously.

7.1 Cumulative Rewards & Cumulative Failures during Training

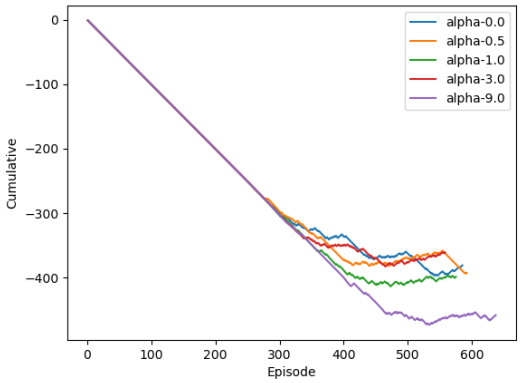

Fig. 8. Cumulative rewards @ minimal

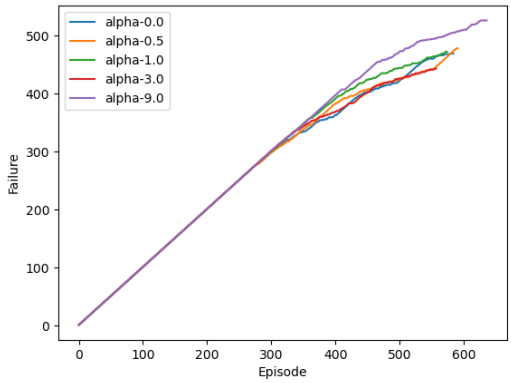

Fig. 9. Cumulative failures @ minimal

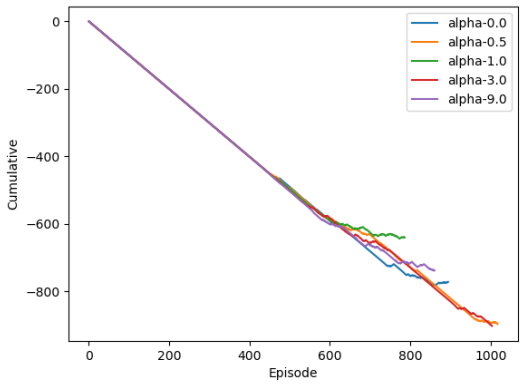

Fig. 10. Cumulative rewards @ intermediate

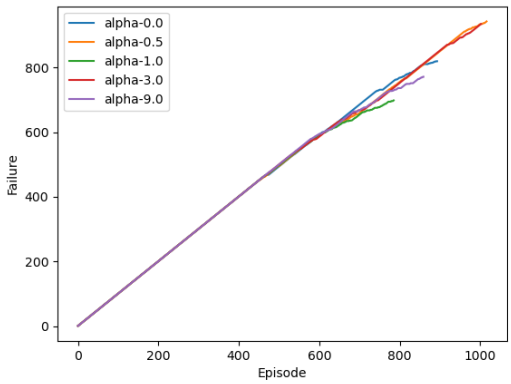

Fig. 11. Cumulative failures @ intermediate

We see an overall similar pattern over our different configurations of obstacles, all alphas start at a similar pace before diverging as to when the agent learns not to crash. The cumulative rewards mostly stabilizes in the minimal setting, where alpha 0.5 is the first to reach a plateau. In the intermediate setting, some alphas reach a plateau much earlier than RL by itself. Precisely, in the intermediate case, alpha 1 is the first to reach a plateau, while alpha 3 and alpha 9 fail to ever reach one. Note that the number of episodes are different for each setting because all have been trained over 250k frames, so the less episodes ran, the quicker the agent learnt to not crash and thus can exhaust its frames 'living' rather than accumulate episodes of failures. In both the minimal setting, the one that has the least number of episodes is 3.0 while in the intermediate case, that is alpha 1.0.

7.2 Failures during Training

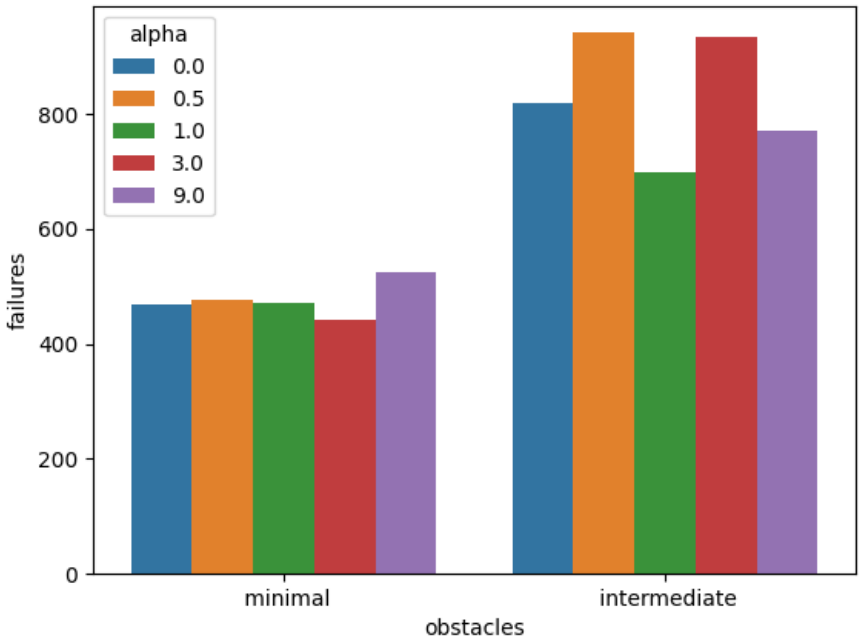

Fig. 12. Overall failures during Training

Although, there is no substantial changes in the overall failures when dealing with minimal obstacles, as the scenario gets more complex, i.e. intermediate number of obstacles in the world, we see that alpha 1 allows the agent to crash less before learning. During training, some alphas allows us to ensure more safety to the agent, as in minimize its crashes. As the agent fails less, it is able to increase its cumulative rewards earlier. We are left with the question of evaluating performance during deployement to see if this added safety takes away from the learned policy, either in terms of average frames to reach the target, or more failures during deployement. We also see that different alphas behave very differently with no apparent evidence as to which is optimal.

7.3 Policy during Deployement

Alphas	Frames	Reward
0.0	962 ± 173	0.80 ± 0.42
0.5	1000 ± 11	0.90 ± 0.00
1.0	764 ± 332	0.1 ± 0.96
3.0	983 ± 110	0.82 ± 0.38
9.0	967 ± 143	0.76 ± 0.50

Table 1. Mean frames and rewards @ minimal

Alphas	Frames	Reward
0.0	872 ± 248	0.41 ± 0.84
0.5	652 ± 366	-0.29 ± 0.95
1.0	872 ± 284	0.39 ± 0.84
3.0	998.9 ± 74	0.84 ± 0.33
9.0	474 ± 368	-0.59 ± 0.81

Table 2. Mean frames and rewards @ intermediate

In the minimal setting, 0.5 has the least variance and the highest mean reward per episode. All alphas have positive mean rewards, meaning they succeed more than fail, although 1.0 has the most variance and is the least performing. For all other alphas, their performance is comparable to RL by itself, 0.5 is the only one that outperforms. In the intermediate setting, 0.5 and 9.0 are the worst performing, averaging a negative reward per episode. RL by itself performs similarly to 1.0 and is outperformed by 3.0 that averages the highest mean reward, and has the most average of steps, meaning it lives the longest.

7.4 Failures during Deployement

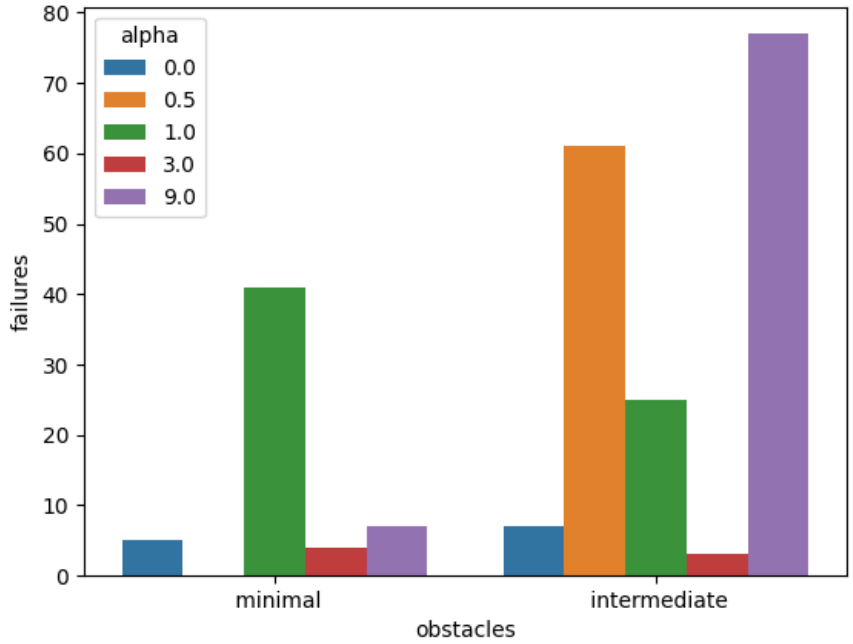

Fig. 13. Overall failures during Deployement

We see that during deployement, the most unsafe policy is 9.0 for the intermediate setting, and 1.0 for the minimal setting, which agrees with our discussion over the deployed policies above. Although in the minimal setting, during training, 9.0 crashes more than all other alphas, it still behaves better than 1.0. On the other hand, 0.5 performs the best across all other alphas in the minimal setting. It is the least uncertain during deployement, and does not crash. During training in the minimal setting, 0.5 was also the first to reach a plateau, meaning it had the most opportunity to learn to minimize crashing. In the intermediate setting, no other alpha is more safe than RL by itself, except 3.0 by a small margin, which also outperforms it in the learned policy when it comes to averaged rewards.

8 DISCUSSION

Our above results show us that there is no apparent difference over alphas during deployement, except that some result in 'worse' safety outcomes, i.e. crashing more than RL by itself. However, during training, several alphas, allow us to minimize failures, and thus might allow us to train agents in the real world where safety is important. For 'good' alphas, there does not seem to be a loss in optimality of policy, rather some allow the agent to behave better than RL by itself, especially given the average reward per episode. To confirm those results, it would be interesting to extrapolate the alphas for more obstacles, where learning has to be done faster OR increase the number of frames to train, to allow all alphas to reach a plateau and thus compare optimality in terms of average frames, i.e. how quickly does the agent reach the goal.

9 CONCLUSIONS

In conclusion, as RL faces the issues of reward shaping, meta-learning and the exploration-exploitation dilemma, domain knowledge might offer a way of improving reinforcement learning methods to ensure better safety. The DIO architecture offers a seamless way to translate from a domain-dependent language, i.e. prolog, to a numerical value that can inform a reinforcement learning agent. Our architecture was tested on a simple gridworld with dynamic obstacles. Our results show that some weights to DIO allowed less crashes during training and resulted in better policies. Although those improvements are minimal, they need further investigation in more complex settings to truly conclude. Those weights do not have an 'apparent' optimal whether in the minimal or intermediate setting, and should be investigated further, as otherwise, they become another hyperparameter that the AI programmer needs to bruteforce and experiment on.

REFERENCES

[1] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. 2017. Constrained Policy Optimization. *Proceedings of the 34th International Conference on Machine Learning (ICML)* (2017).

[2] Tarek R. Besold, A. Garcez, Sebastian Bader, H. Bowman, Pedro M. Domingos, P. Hitzler, Kai-Uwe Kühnberger, L. Lamb, Daniel Lowd, P. Lima, L. Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. 2017. Neural-Symbolic Learning and Reasoning: A Survey and Interpretation. *ArXiv* abs/1711.03902 (2017).

[3] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. 2018. Minimalistic Gridworld Environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid.

[4] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. *IJCAI*, 2462–2467.

[5] Kurt Driessens. 2010. *Relational Reinforcement Learning*. Springer US, Boston, MA, 857–862. https://doi.org/10.1007/978-0-387-30164-8_721

[6] DAAN FIERENS, GUY VAN DEN BROECK, JORIS RENKENS, DIMITAR SHTERIONOV, BERND GUTMANN, INGO THON, GERDA JANSSENS, and LUC DE RAEDT. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015), 358–401. https://doi.org/10.1017/S1471068414000076

[7] Jakob N. Foerster, Yannis M. Assael, N. D. Freitas, and S. Whiteson. 2016. Learning to Communicate to Solve Riddles with Deep Distributed Recurrent Q-Networks. *ArXiv* abs/1602.02672 (2016).

[8] Marek Grzes. 2010. Improving Exploration in Reinforcement Learning through Domain Knowledge and Parameter Analysis. (03 2010).

[9] Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, and Sergey Levine. 2018. Meta-Reinforcement Learning of Structured Exploration Strategies. In *Conference and Workshop on Neural Information Processing Systems (NeurIPS)*. 10.

[10] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. 2022. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research* 73 (2022), 173–208.

[11] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Intell. Res.* 4 (1996), 237–285.

[12] Andrej Karpathy and Fei Li. 2015. Deep visual-semantic alignments for generating image descriptions. 3128–3137. https://doi.org/10.1109/CVPR.2015.7298932

[13] Jens Kober, J. Bagnell, and Jan Peters. 2013. Reinforcement Learning in Robotics: A Survey. *The International Journal of Robotics Research* 32 (09 2013), 1238–1274. https://doi.org/10.1177/0278364913495721

[14] Sven Koenig and Reid G. Simmons. 1996. The Effect of Representation and Knowledge on Goal-Directed Exploration with Reinforcement-Learning Algorithms. *Machine Learning* 22, 1 (01 Jan 1996), 227–250. https://doi.org/10.1023/A:1018068507504

[15] Adam Laud. 2011. Theory and Application of Reward Shaping in Reinforcement Learning. (04 2011).

[16] Yuxi Li. 2019. *Reinforcement Learning Applications*. Technical Report arXiv:1908.06973. http://arxiv.org/abs/1908.06973

[17] Scott E. Reed, Yi Zhang, Y. Zhang, and Honglak Lee. 2015. Deep Visual Analogy-Making. In *NIPS*.

[18] Jesus Romero-Hdz, Baidya Nath Saha, Seiichiro Tstutsumi, and Riccardo Fincato. 2020. Incorporating domain knowledge into reinforcement learning to expedite welding sequence optimization. *Engineering Applications of Artificial Intelligence* 91 (2020), 103612. https://doi.org/10.1016/j.engappai.2020.103612

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

[20] Nicolas Schweighofer and Kenji Doya. 2003. Meta-learning in Reinforcement Learning. *Neural Networks* 16, 1 (Jan. 2003), 5–9.

[21] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[22] Ana Tenorio-González, Eduardo Morales, and Luis Villaseñor-Pineda. 2010. Dynamic Reward Shaping: Training a Robot by Voice. 483–492. https://doi.org/10.1007/978-3-642-16952-6_49

[23] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. 3156–3164. https://doi.org/10.1109/CVPR.2015.7298935