SENIOR THESIS 2021-22

Domained Informed Oracle (DIO) in Reinforcement Learning

Samar Rahmouni
srahmoun@andrew.cmu.edu

Advisor: Prof. Giselle Reis
giselle@cmu.edu

Abstract

Reinforcement learning (RL) is a powerful AI method 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 current state configurations to a numerical value. The agent's goal is then to maximize its cumulative reward over its lifetime. Unfortunately, there is no systematic method to design a reward function. This needs to be done on a case by case basis, and might be hard depending on how the states are represented. States are typically represented as vector of values in RL, and translating properties and rules from a domain into this representation can be complicated depending on how many values are used, what they represent, whether they are normalized or not, etc.

We propose a *Domain Informed Oracle (DIO)* as a solution for systematically incorporating 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 specific knowledge using an expressive and intuitive language, where they can define states and rules in the most convenient way. DIO provides an informed decision to the reward function, thus allowing it to dynamically adapt the rewards.

Our implementation is tested on a Traffic Simulator scenario and compared to a basic uninformed RL algorithm. The comparison is based on performance which we define by three metrics: time to train, optimality of the learned policy and finally, number of errors states reached.

1 Introduction

Implementing a robust adaptive controller that is effective in terms of precision, time, and quality of decision when facing dynamic and uncertain scenarios, has always been a central challenge in AI and robotics. Precisely, as we want our autonomous agents to be 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 their optimality and time taken to produce a decision. As autonomous cars are deployed, IoT is popularized, and human-robot interactions become more complex, we are more and more confronted with the need for robotic agents that can effectively and continually adapt to their surroundings, not only in simulation, but also in practice, when deployed as a cyber-physical system. Since we are unable to provide a repertoire of all possible scenarios and actions, our agents need to be able to autonomously predict and adapt to new changes. Reinforcement Learning (RL) is an approach that supports dynamically adapting to new input. It is also the solution that AlphaGo, Deepmind AlphaStar, and OpenAI Five have adopted [1], respectively for Go, StarCraft II and Dota 2 and found success in.

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 a trial-and-error process guided by a *domain dependent* reward function. For example, if the agent is a self-driving car, the reward function can greatly penalize states when 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 a reward function that ensures optimality is thus crucial. 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. We argue that such a combination will ensure a faster and more efficient RL trained agent in terms of optimality. The proposed combination is tested on a traffic simulation and the results are compared with a RL implementation that makes use of standard practices to design a reward function.

2 Reinforcement Learning

Reinforcement Learning is a method of learning that maps situations to actions in order to maximize its rewards [2]. 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. Note that a 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. Consider Figure 1.

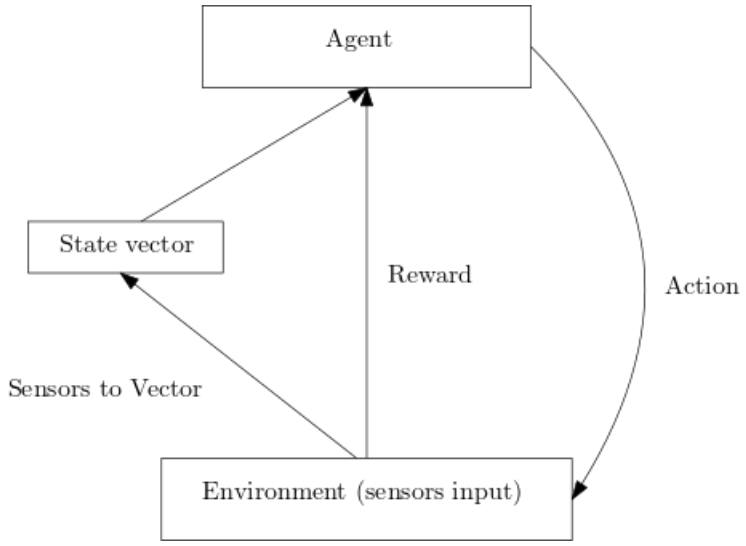

Figure 1: Reinforcement Learning Routine

The diagram in Figure 1 is a high-level description of how an agent using reinforcement learning can be trained. The lower 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 state is. The agent's goal is to increase its reward by taking actions that reach better states each time.

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 the right reward function, we can ensure a faster and more efficient training, possibly with fewer errors.

2.1 Challenges in Reinforcement Learning

Reward shaping (1), the exploration-exploitation dilemma (2) and meta-learning (3) are main challenges that make it harder for RL to be adopted as a solution to more real-world problems.

Reward shaping [3] refers to the lack of systematic methods to design a reward function that ensures fast and efficient learning. This generation of an appropriate reward function for a given problem is still an open challenge [4]. 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 [5]. However, most RL agents can only stay in simulation due to the lack of safety guarantees. This is because of the trial-and-error nature of the RL training. Thus, there exists a need for *shaping rewards* instead. There are reasonable criteria on how this should be done, those are *standard practices*. For instance, rewards that can be continuously harvested speed up convergence compared to rewards that can only be harvested at "the end" (*i.e.* the chess example). Similarly, one should avoid only negative rewards as that results in unwanted behavior. Furthermore, if dealing with a continuous state space, it helps to have a polynomial differential function as the reward function as it is shown to help the agent learn faster. Finally, one can normalize rewards at the end as to not end up with too many discrepancies. However, there still exists a lack of a systematic method to design a reward function, and this needs to be done on a case by case basis. Moreover, the abstract representation of states and actions can make it difficult to map domain specific rules and judgments to the reward function. A consequence of finding a systematic method to shaping rewards handles (2).

The *exploration vs. exploitation dilemma* is the question of whether to always exploit what the agent knows or explore in the hope that an unexplored state might result in better rewards. This dilemma of *exploration* vs. *exploitation* is a central issue of RL. Consider this problem. An agent is at an intersection. It has the choice of going either right or left. It does not yet know the outcome of either. It chooses right at a given point and receives a reward $r = 1$. The question is "When faced with the same decision, should it keep going right?" There are two issues to consider. First, it does not know the outcome of going left. It could be that there is a better reward waiting for it on the left lane. Second, when dealing with a stochastic environment, it might be that r was a one-time occurence. It would be equivalent to someone buying a lottery ticket, and winning \$1M on their first try, and thus, spending all that they won in trying to make it happen again. This problem showcases the importance of exploration; an agent needs to see where other paths might lead to, but also exploitation; if it keeps exploring forever it will never accumulate rewards. This is especially evident when the possible states cannot be exhausted. Several techniques have been proposed to balance between exploration and exploitation [6]. A notable one is the *epsilon-greedy* technique. The idea is to set some probability ε by which the agent decides to explore. This probability can be adapted to decrease as more *episodes* are completed. However, by ensuring (1), an informed reward function is able to sufficiently deter the exploration of undesirable states while encourage the exploitation of desirable ones, continuously adapting to acquired knowledge and resolving the conflict when necessary. More interestingly, a solution to (2) impacts (3).

Meta-learning is the problem of deploying an agent trained in a simulation to the real-world, or possibly another simulation, where it encounters state configurations it did not during its training. The goal is to be able to efficiently adapt to those configurations. The problem of meta-learning in RL stems from the uncertainties of the world. Consider the result of training: It is a policy π, a function to map states to action $\pi : S \rightarrow A$. The learned policy is the one that maximizes the cumulative rewards. This training is most often done in simulation, given the lack of safety guarantees of RL. However, several problems come into place when considering the deployment of the trained agent. Considering that an agent has done well in a designated simulation does not imply that it will do as well in the real-world. Overall,

it must be that certain uncertainties will not be expected, thus there can be no expectation on how the agent will behave when out of simulation. Meta-learning in reinforcement learning is the problem of learning-to-learn, which is about efficiently adapting a learned policy to conditions and tasks that were not encountered in the past. In RL, meta-learning involves adapting the learning parameters, balancing exploration and exploitation to direct the agent interaction [7, 8]. Meta-learning is a central problem in AI, since an agent that can solve more and more problems it has not seen before, approaches the ideal of a general-purpose AI. However, as noted previously, a solution to (2) implies a continuous adaptation to knowledge. Since the conflict of exploration and exploitation is resolved, the agent adapts accordingly to tasks it encountered in the past (exploiting), but also tasks it encounters for the first time (exploring). Thus, from (2) one can have a significant impact on (3).

3 Symbolic Reasoning for Reinforcement Learning

To tackle the challenges 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 of DL on its lack of formal semantics and intuitive explanation, and the lack of expert knowledge towards guiding machine learning models. A key question the field targets is identifying the necessary and sufficient building blocks of AI [9], namely, how can we provide the semantics of knowledge, and work towards meta-learning? 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. [10] This leads to neurosymbolic AI finding various applications including vision-based tasks such as semantic labeling [11, 12], vision analogy-making [13], or learning communication protocols [14]. Those results inspire us to use those techniques for reinforcement learning, as to tackle its challenges.

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)*.

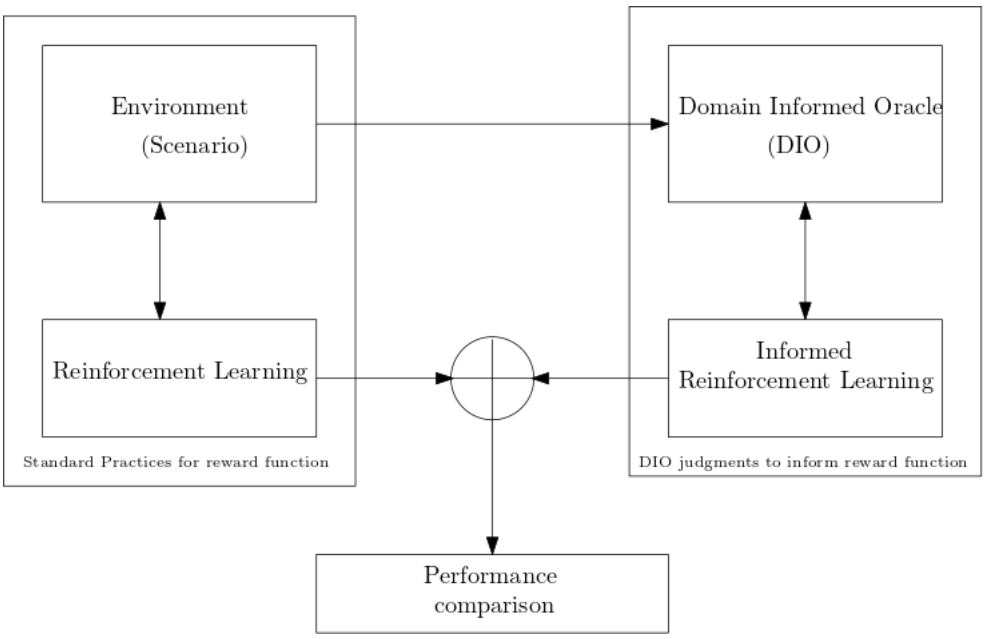

Figure 2: Overview of the proposed solution

The diagram in Figure 2 is a high-level description of our proposed solution. The box on the left represents a basic reinforcement learning algorithm that depends on the scenario of the given problem and the common standard practices discussed previously to design a reward function. The box on the right represents our proposed domain knowledge to inform the reinforcement learning algorithm. Precisely, the domain informed oracle is given the scenario and can thus start a feedback loop between itself and the informed RL module to update the rewards. Finally, those two implementations will be compared based on their performance. In the following, we define performance given three metrics: (1) time to train (2) optimality of the learned policy and (3) number of errors through training. Consider 'errors' as suboptimal decisions that were made by the agent while in the process of training. For example, exploring a (state, action) pair that has previously given a negative reward is suboptimal.

4 Domain Informed Oracle

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 Q-learning, a specific method to compute the policy in RL. It does not mean that our solution is specific to Q-learning.

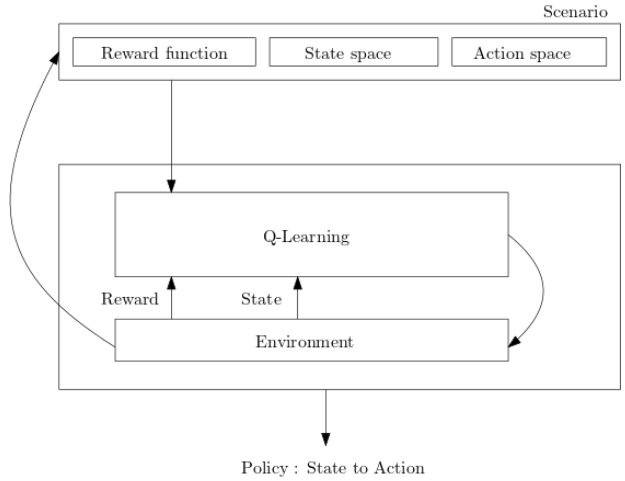

Figure 3: Reinforcement learning architecture

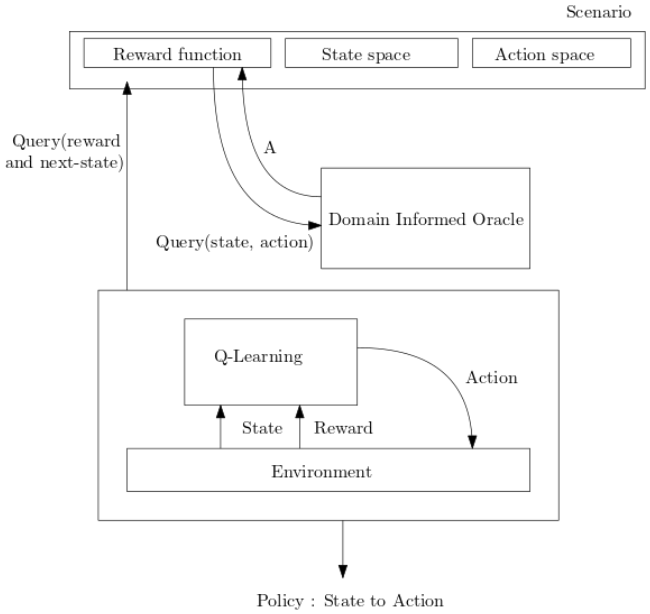

Figure 4: Dio+RL architecture

The diagram in Figure 3 describes the basic routine of RL in more details. The environment defined by the scenario sends the current state to the Q-Learning 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 which we keep obscure. The fundamental idea is that A is used to inform the reward function when it is tasked with computing the reward.

4.2 Specifications

DIO is a logic programming based module that takes the query from the reward function and returns A, a judgment that the scenario awaits to update its reward function. The judgment is obscured as it is a choice of the scenario, independent of the DIO implementation. In the following, we will define A by its behavior, rather than its type. To do so, we will walk through the routine in the diagram of Figure 5.

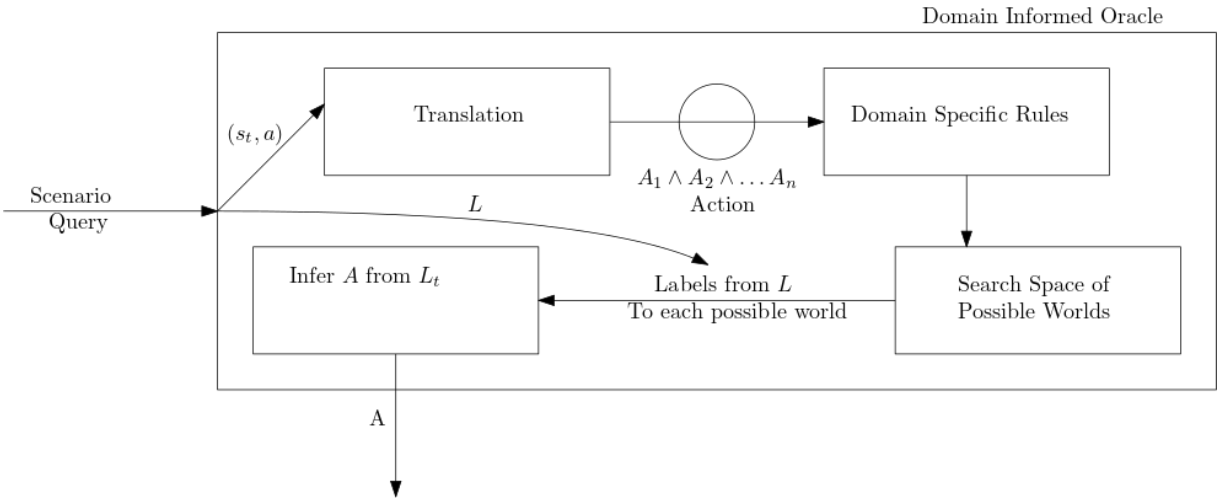

Figure 5: Domain Informed Oracle routine

1. The scenario queries DIO. It sends (s_t, a), the state at time t and the action. It also sends $L : s \rightarrow t$ where t is a sum of types: $t \equiv t_1 + t_2 .. + t_n$. For instance, consider our previous example of the chess game and let the scenario define good openings as "desirable" and random openings as "undesirable". Thus $t \equiv d + ud$ where d is equivalent to desirable and ud to undesirable.

2. The first step of DIO is a translation $T : \mathbb{R}^n \rightarrow t_1 \times t_2 \times ...t_n$ that takes in the state vector s_t and returns a conjunction of propositions $P = A_1 \wedge A_2 \wedge ... \wedge A_n$. This is our set of ground facts.

3. P is passed alongside the action to the domain specific rules defined in DIO.

4. Using step semantics, DIO generates possible worlds up to a given time t'. Note that given the stochasticity of the environment, there is not certainty on whether the worlds expected by DIO will necessarily happen. Consider Figure 6.

5. Those possible worlds, equivalent to states, can be transformed using L to t.

6. At the last step, DIO ends up with a set S of t that defines the labels of all the possible worlds.

7. The last step is left as an inference depending on how the the judgment for deciding a final A from S is formulated. For instance, we can consider a rule where if most of the possible worlds are undesirable, then let A be undesirable.

8. Finally, A is passed to the scenario. Note then that $A : t$.

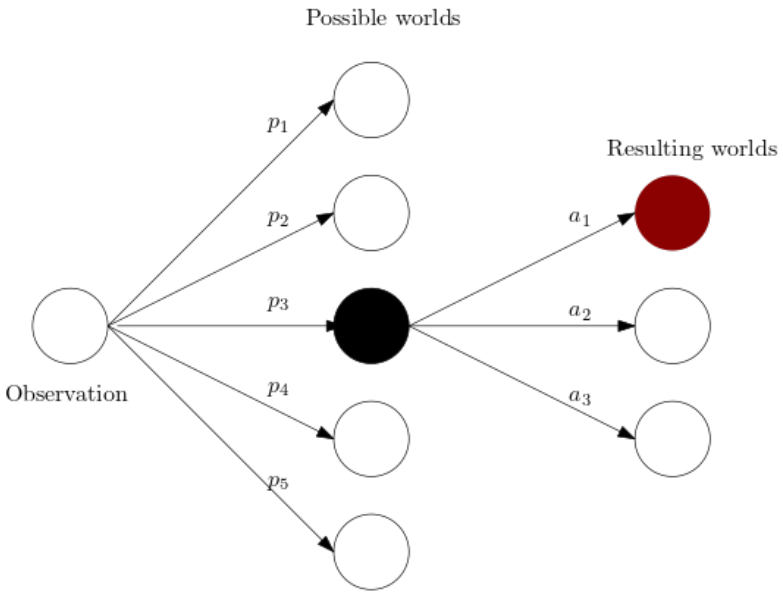

Figure 6: Game Tree Simulation by DIO

Observe how this does not guarantee that the reward function will be informed in a meaningful way. To that end, we need to consider (1) what are the specifications of a 'good' reward function, (2) what conditions should be set on the domain specific rules and L provided by the scenario, (3) how to define a systematic way for the scenario to use this information. Those considerations are left for future investigations.

4.3 Modules Interactions

In practice, we consider the following modules and their interactions as shown in 7.

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. **World 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 predicate, e.g. *crash :- obs(X,Y), agent(X,Y)*. Those labels have probabilities associated with them.

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.

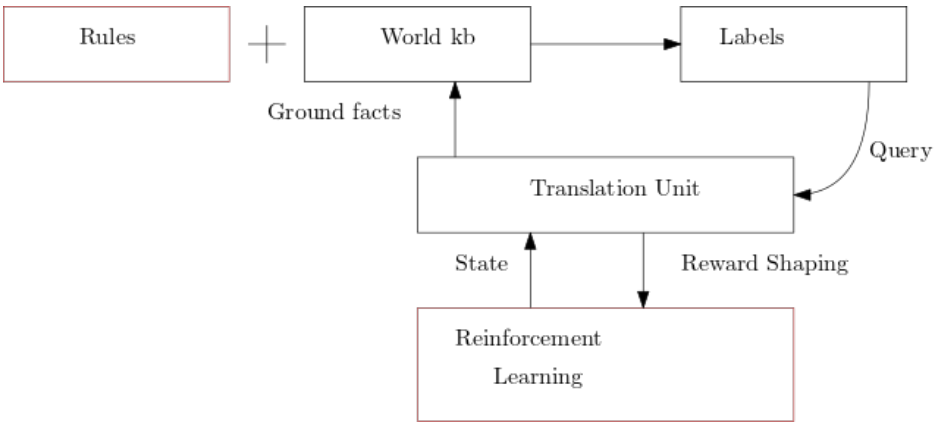

Figure 7: Modules & Interactions

5 Dynamic Obstacles in a GridWorld

We first evaluate the performance of our dio/rl implementation compared to an implementation making only use of rl.

5.1 Scenario in Reinforcement Learning

Our autonomous agent exists in an 8x8 grid world. Its goal is to reach the goal from his initial position (1,1). Along the way, there exists dynamic obstacles which movements is unknown. The agent is punished if colliding with an obstacle and the episode, hereby ends. This environment offered by gym-gridworld [15] 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. The action space is,

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

Finally, the reward is a function of the distance from the goal defined as,

$$R_t = 1 - 0.9 * (\frac{steps}{max\_steps})$$

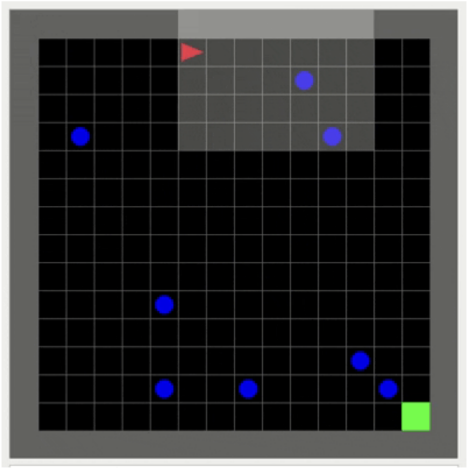

Figure 8: Gridworld with Dynamic Obstacles

The Reinforcement Learning experiments have been performed on 1M frames for a similar start configuration as shown in 8. The episode ends when the agent reaches the goal OR collides with an obstacle. We want to encourage the shortest and safest path, thus, the punishment for crashing is $r = -1$. Our rewards are normalized as shown in the reward function. We define the range of rewards to be $(-1, 1)$.

5.2 Domain Specific Rules

The rules are defined as a ProbLog [16]: a probabilistic prolog that allows us to capture the stochasticity of the environment. 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 take the following form:

$$\frac{P_0 :: \varphi(0) \qquad P_1 :: \varphi(1) \qquad P_2 :: \varphi(2) \qquad P_3 :: \varphi(3) \qquad \dots}{p_1, \dots, p_n} \text{ (action)}$$

We define $\sum_{i=0}^{n} P(i) = 1$, and $\varphi(i)$ corresponds to the conjunction of grounds facts of the possible world with probability P_i. The action is equivalent to our step semantics, thus, we enforce that a given action modifies the facts in some form. In practice, an action is the missing clause to generate the next predicate. In the gridworld example, we give the following.

$$(1)\ \frac{\text{atPos(X + V*T, Y)}}{\text{atPos(X,Y), speed(V), timestep(T)}}\ \text{(right)} \qquad (2)\ \frac{\text{0.25 :: obs(X + V*T, Y, V) ...}}{\text{obs(X,Y,V), timestep(T)}}\ \text{(time)}$$

(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. We could do the same for (1) by consider 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,

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

5.4 From Norms to Labels

Todo.

5.5 Translation Unit

6 Related Work

As discussed previously in Section 3, 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 [17, 18, 19, 20]. [20, 18] use the general idea of *reward shaping* and *epsilon adaptation* respectively to incorporate procedural knowledge into a RL algorithm. Both introduce this combination as a successful strategy to guide the exploration and exploitation tradeoff in RL. They both show promising results. While [20] 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 [17]. Precisely, in [17], 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 be potentially offer

a solution to the problem of meta-learning. While [18, 17] both present promising rewards, they lack the modularity necessary for scaling the proposed methods to further RL implementations. Those by taking their results into consideration, we are hopeful that symbolic reasoning and RL integration could provide a solution for reward shaping, meta-learning and the exploration-exploitation dilemma.

7 Conclusions

In conclusion, as RL faces the issues of reward shaping, meta-learning and the exploration-exploitation dilemma, domain knowledge show promising results in improving reinforcement learning methods. The main challenge is to make such an integration seamless, and independent of the AI implementation. As we start incorporating our modules into one architecture to test on our Traffic Simulator in Section **??**, we will be able to show preliminary results. Similarly, we look further into the specifications of DIO to ensure that it informs the scenario in a meaningful way as discussed in Section 4.2.

References

[1] Yuxi Li. Reinforcement Learning Applications. Technical Report arXiv:1908.06973, August 2019.

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

[3] Adam Laud. Theory and application of reward shaping in reinforcement learning. 04 2011.

[4] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013.

[5] Ana Tenorio-González, Eduardo Morales, and Luis Villaseñor-Pineda. Dynamic reward shaping: Training a robot by voice. pages 483–492, 11 2010.

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

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

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

[9] Artur d'Avila Garcez and Luis C. Lamb. Neurosymbolic ai: The 3rd wave, 2020.

[10] 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. Neural-symbolic learning and reasoning: A survey and interpretation. *ArXiv*, abs/1711.03902, 2017.

[11] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. pages 3156–3164, 06 2015.

[12] Andrej Karpathy and Fei Li. Deep visual-semantic alignments for generating image descriptions. pages 3128–3137, 06 2015.

[13] Scott E. Reed, Yi Zhang, Y. Zhang, and Honglak Lee. Deep visual analogy-making. In *NIPS*, 2015.

[14] Jakob N. Foerster, Yannis M. Assael, N. D. Freitas, and S. Whiteson. Learning to communicate to solve riddles with deep distributed recurrent q-networks. *ArXiv*, abs/1602.02672, 2016.

[15] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018.

[16] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. pages 2462–2467, 01 2007.

[17] Kurt Driessens. *Relational Reinforcement Learning*, pages 857–862. Springer US, Boston, MA, 2010.

[18] Jesus Romero-Hdz, Baidya Nath Saha, Seiichiro Tstutsumi, and Riccardo Fincato. Incorporating domain knowledge into reinforcement learning to expedite welding sequence optimization. *Engineering Applications of Artificial Intelligence*, 91:103612, 2020.

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

[20] Marek Grzes. Improving exploration in reinforcement learning through domain knowledge and parameter analysis. 03 2010.

A Appendix

In the following, we describe our previous work that was done towards considering safe reinforcement learning. Precisely, we were motivated by the lack of safety guarantees and the use of symbolic reasoning towards incorporating safety properties from a verified safe controller (SC) i.e. a declarative language module to compute safe and unsafe states. Those safety states are computed during training, thus ensuring that no unsafe states are explored by the RL. The general idea was to allow training in the real-world, thus taking away the need for simulation. Our proposed RL+SC architecture is shown in Figure 9.

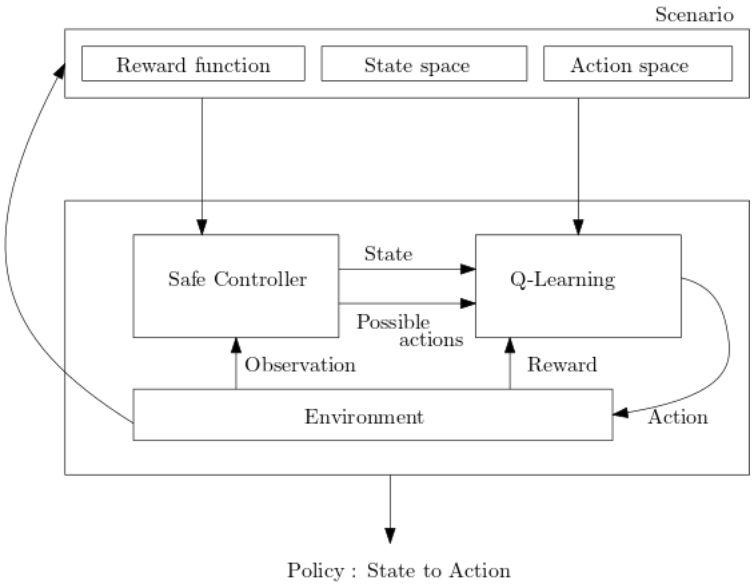

Figure 9: Overview of the proposed solution

1. The environment sends the observation to the SC.
2. The SC computes the state from the observation, thus keeping only ground facts that uphold the *Markov Property* (see Definition 9.1).
3. The SC also computes the set of possible actions A that ensure that no unsafe states is reached. This is done by searching for possible configurations that follow a (state, action) pair over all possible actions from the action spaces.
4. The state and set of possible actions is sent to the RL algorithm, thus only allowing the agent to choose from the set actions.
5. Next steps follow from the RL basic routine from Figure 1

The implementation can be found in the thesis corresponding github under the rl-implementation folder[1]. For our case study, we chose the vehicle platooning problem. The setting is as follows; two vehicles, one leader and one follower. Their goal is to minimize the gap between them while ensuring that they do not crash.

We then present our preliminary results before arguing for the issues with our approach.

[1] https://github.com/natvern/Thesis

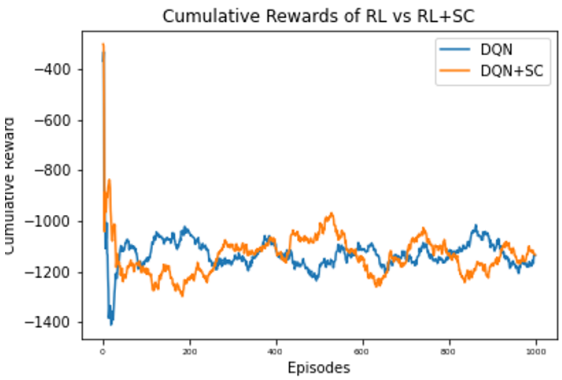

Figure 10: Cumulative Rewards of RL vs. RL+SC

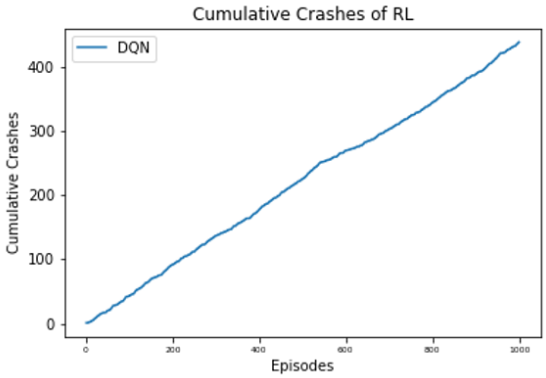

Figure 11: Cumulative Crashes of RL during Training

The graph in Figure 10 shows the cumulative rewards of the basic RL implementation without the safe controller (blue line) vs. the RL+SC implementation (orange line). As evident in the cumulative rewards, all are negative. This is because of our reward function choice that only punishes for distances not equal to the desired gap. Thus, learning is not optimal. We can however see that the cumulative rewards of both implementation does not change by a significant amount. The graph in Figure 11 shows the number of crashes of the basic RL during training. We can easily deduce from the linear graph that the RL without the SC never learns not to crash.

Definition A.1 (Markov Property)**.** The next state only depends on the value of the current state. In other words, only the present can affect the future. In terms of our state vector, this means that given S_t, I only need the features in S_t to predict S_{t+1}.

A.1 Issues with the safe RL approach

- The simplicity of our problem setting does not showcase the need for a SC as we only consider two vehicles on an x-axis.
- Though ensuring safety guarantees during training theoretically takes away the need for a simulation, in practice, the uncertainties of the real world cannot be all predicted. This results in a handful of safety properties that might be guaranteed, not enough to allow sensitive cyber-physical systems to be trained using RL in the real world.
- If an oracle existed that was able to predict all uncertainties of the real world, this oracle could then be used to solve the optimization problem deterministically without the need for RL.
- Depending on the SC implementation, safety guarantees can be too strict thus completely destroying optimality. Consider a SC that only allows a car not to move. Though it guarantees that no crashes happen, it is also too strict, thus going against the goal of the agent.