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 grid worl with dynamic obstacles and later extended to a traffic simulation scenario. It is then 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, the success percentage, i.e. the number of times it reaches a positive terminal state, a goal for example.

Our results show that although the time to train is longer, the learned policy using DIO succeeds in 90% of the time to reach the goal. The policy both incorporates safety by avoiding crashing into an obstacle but also optimality by choosing the fastest possible path. This is not the case with an implementation that only makes use of RL, precisely a proximal policy optimization that does learn safety but always ends its episode in the maximum number of steps, rather than reaching the goal.

Contents

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 $R : (S \times A) \to \mathbb{R}$ 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.

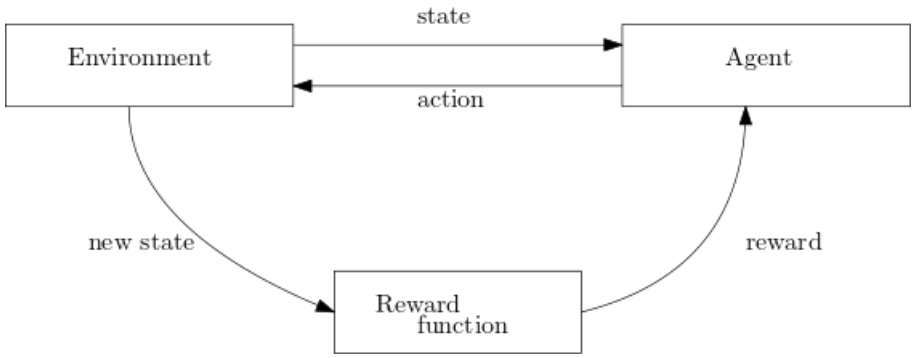

Figure 1: Reinforcement Learning Routine

The diagram in Figure 1 is a high-level description of how an agent using reinforcement learning can be trained. The 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 the right reward function, we can ensure a faster and more efficient training, possibly with fewer errors.

For more information on the algorithms we have used for our implementations, refer to A.2.

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. 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. It provides a more declarative approach to reason about rewards, thus providing a systematic

method to map labels to rewards. As a consequence, it is able to handle (2), namely, the exploitation vs. exploration dilemma.

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: a function π that map states to actions $\pi : S \rightarrow A$. 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]. As a result, neurosymbolic AI has been successfully applied to 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.

Specifcally, 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).

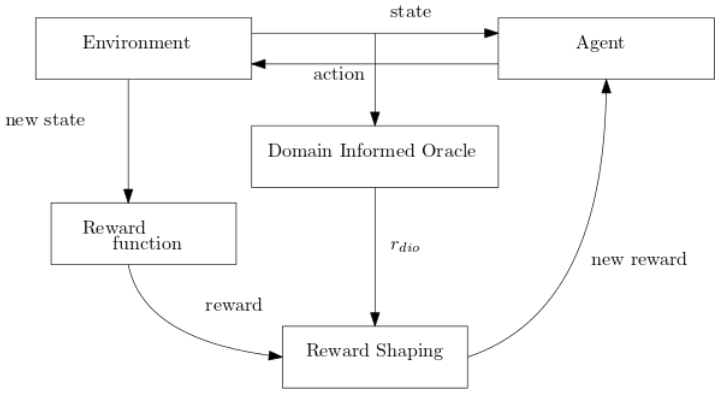

Figure 2: Overview of the proposed solution

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.

Those two rewards are merged by the reward shaping unit. We could either overwrite the initial reward and only trust DIO, likewise we could trust them both equally and take their average. This is left as a *design choice*. Finally, this computed new reward is given to the agent that is unaware of this new mechanic.

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 Proximal Policy Optimization, a specific method to compute the policy in RL that is explained in more details in A.2. It does not mean that our solution is specific to it, rather it can be generalized to any algorithm choice.

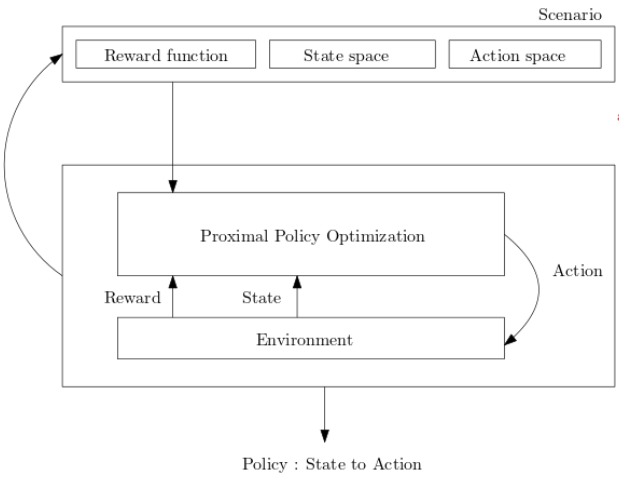

Figure 3: Reinforcement learning architecture

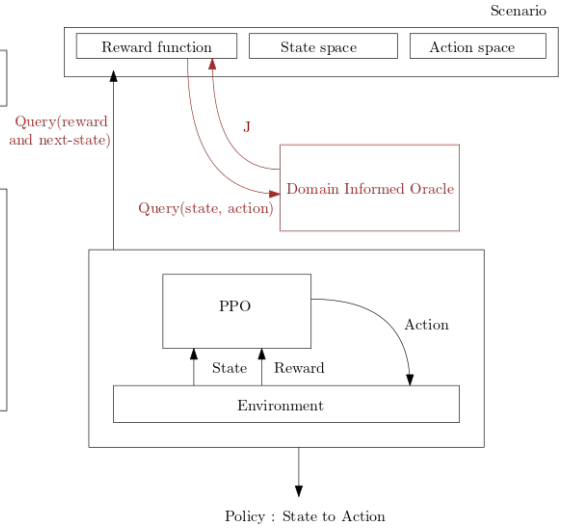

Figure 4: DIO+RL architecture

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 *Proximal Policy Optimization* algorithm. More information on the algorithms used found in A.2. 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 J, a judgment which we keep obscure. The fundamental idea is that J is used to inform the reward function when it is tasked with computing the reward.

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 *maybe-crash :- nextObs(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.

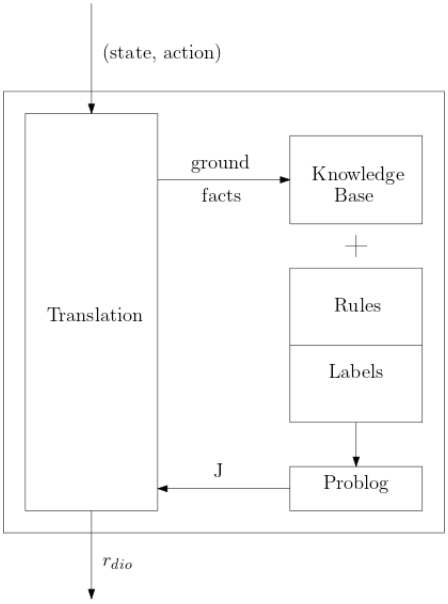

Figure 5: Modules & Interactions

Figure 5 describes the interactions of the different modules, basically taking a closure look to DIO. 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'. This new reward can either *overwrite* the previous reward, or *fine-tune it*. In general, the way this reward affects the initial reward is a **design choice** that we leave for future investigations. Finally, as a result of the RL action, the next state is given to TU that translates it into Ground facts to update the world knowledge, thus restarting the loop. 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 [15]. 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 is a subset of ground probabilistic facts where the probability of the subset is the product of the probabilities of the facts it contains.

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.

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.

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.

5 Dynamic Obstacles in a GridWorld

We first evaluate the performance of our DIO/RL implementation compared to an implementation making only use of reinforcement learning. We precisely look at (1) frames per second to evaluate speed. A frame is a the step taken by the agent at one unit of time. Thus, frames/second during training are equivalent to how fast the learning takes. We also look at the mean of frames per episode and the mean of returns per episode to evaluate optimality. An episode is defined as the collection of steps before we reach a terminal state, either through fatalities (e.g. colliding with an obstacle), goals or reaching the maximum number of steps that we set to 256 steps. Finally, returns is equivalent to a discounted summation of rewards.

5.1 Scenario in Reinforcement Learning

Our autonomous agent exists in an 8x8 grid world. Its goal is to reach the lower right position (8,8) from his initial position (1,1). Along the way, there exists dynamic obstacles whose movements is unknown. The agent is punished if colliding with an obstacle and the episode, hereby ends. This environment offered by gym-gridworld [16] 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,

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

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

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

Note that the max_steps is defined by the manhattan distance between the agent and the goal. The intuition is we want to reward the agent the faster it reaches the goal. Additionally, the agent is punished with -1 if it collides with an obstacle, to encourage staying away from the dynamic obstacles.

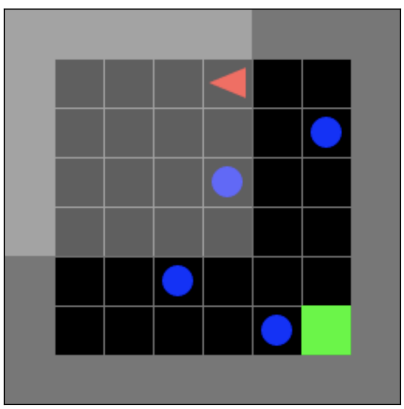

Figure 6: Gridworld with Dynamic Obstacles

The Reinforcement Learning experiments have been performed on 80k episodes for a similar start configuration as shown in Figure 6. The episode ends when the agent reaches the goal **or** collides with an obstacle **or** reaches the maximum number of steps which is 256. We want to encourage the shortest and safest path, thus, the punishment for crashing is $r = -1$.

5.2 Domain Specific Rules

The rules are defined as a ProbLog [17]: a probabilistic prolog that allows us to capture the stochasticity of the environment, as we previously introduced in 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:

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

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, 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.

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

(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 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,

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

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 three possibles norms, (1) *crash* when the agent and the obstacle overlap, (2) *maybecrash* when the agent and the obstacle are one square from each other and finally (3) *safe* when there is ample distance between the agent the obstacle in this case, two square distances. Those norms are identified as 'labels', i.e. predicates defined as follows.

$$\frac{1 :: atPos(X,Y) \quad 1 :: obs(X,Y,\_)}{1 :: crash} \qquad \frac{P_1 :: atPos(X,Y), P_2 :: obs(X,Y,\_) \quad \dots}{P_1 \times P_2 :: maybecrash}$$

$$\frac{P_1 :: atPos(X_1,Y_1) \quad P_2 :: obs(X_2,Y_2,\_) \quad X_1 \neq X_2 \quad Y_1 \neq Y_2 \quad \dots}{P_1 \times P_2 :: safe}$$

$$\frac{\_ : nextPos(X,Y) \quad \_ : goal(I,J) \quad \_ : time(T) \quad L = abs(I-X) + abs(Y-J)}{1.0/(L+T) :: close}$$

Nondeterminism If we look at crash and maybecrash predicates, we notice that the two hold for the same premises. This is what we define as non-determinism, where any of the two can be chosen. The choice was made considering that a crash would imply a 'maybecrash', thus trivially true.

5.5 Translation Unit

The translation unit is the bridge between DIO and RL. Precisely, 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, reward = Step'(S_t, ac)                        ▷ Call to Initial Env
8:      J = getFeedback()                                   ▷ Query DIO
9:      reward = TU.getReward(J)
10:     return (obs,reward)
```

When it comes to reward shaping, we focus on the 'getFeedback' function. First, the translation units gets the labels, i.e. the probabilities of the evaluated predicates and constructs a function that associates every predicate to its corresponding numerical value. Finally, the reward is equivalent to weighted probabilities as shown in Algorithm 2, this is because of our *design choice* to make our judgment as a numerical value. Trivially, the GETREWARD from TU is an identity function.

Algorithm 2 Inference of Judgment J

```
1:  procedure GETFEEDBACK
2:      labels ← getLabels()          ▷ Evaluated predicates w/ Corresponding Probabilities
3:      P ← {crash, maybecrash, safe}                       ▷ Set of possible worlds
4:      F ← {−1, −0.5, 1}                                   ▷ Numerical value 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 weighted probabilities
9:          i ← i + 1
```

5.6 Results

Our testing was done on a Dynamic Obstacles grid world using proximal policy optimization over 80k episodes. We pick up the frames per second to evaluate the speed of learning, the mean number of frames per episode to evaluate the optimality of the solution and finally, the mean returns per episode to evaluate convergence relative to the number of frames. The code is available on github [18] with steps on how to replicate our model. Refer to the beginning of 5 for an explanation of the metrics.

We first visualized the learned policy in the two implementations. Although both learned how to not crash into the Dynamic Obstacles, the implementation with DIO was better at minimizing the number of steps but also learned to reach the goal.

First, as expected, the implementation using DIO is much slower only able to run an average of 150 episodes per second. As a result, for 80k episodes, the implementation with DIO took 9 minutes to train versus 16 seconds for the implementation without. However, this loss in time shows as in 13 a gain in

DIO + RL

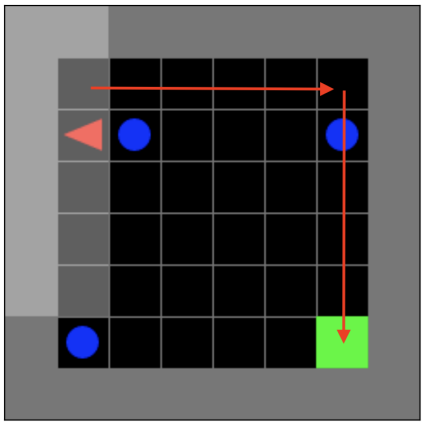

Figure 7: Policy learned with DIO

RL

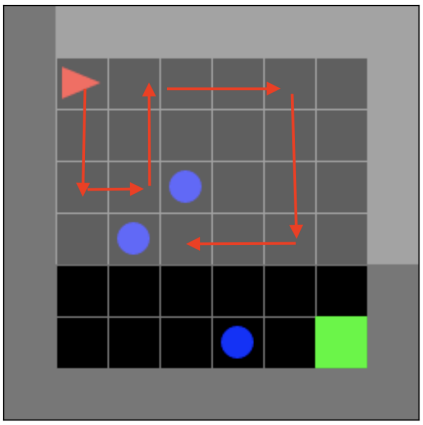

Figure 8: Policy learned without DIO

Figure 9: Frames per Second with DIO

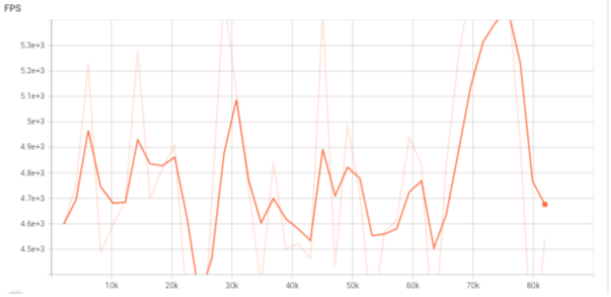

Figure 10: Frames per Second without DIO

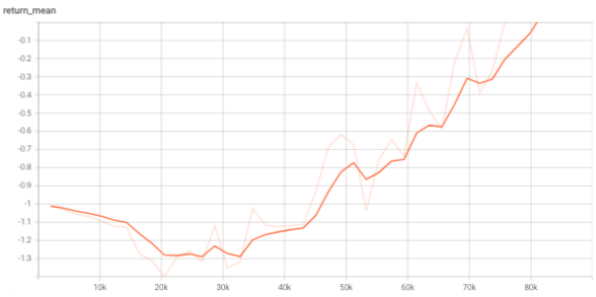

Figure 11: Mean of returns per episode with DIO

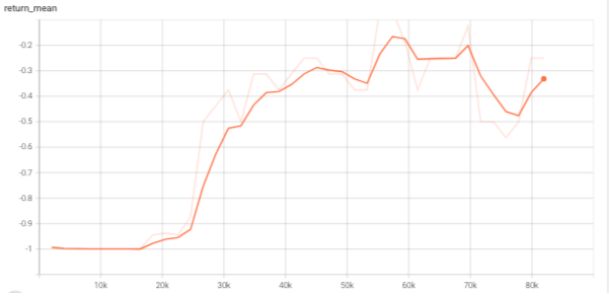

Figure 12: Mean of returns per episode without DIO

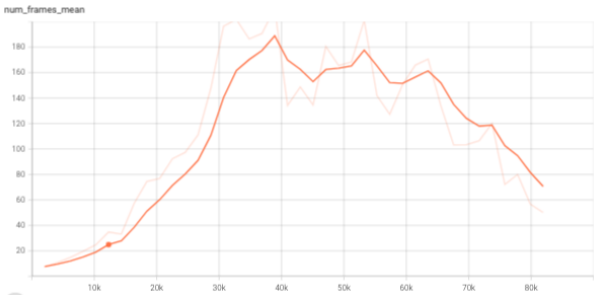

Figure 13: Mean of frames per episode with DIO

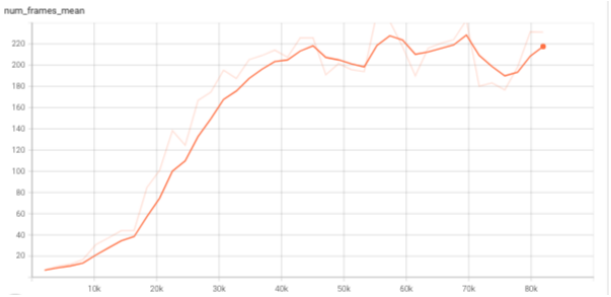

Figure 14: Mean of frames per episode without DIO

optimality. When visualizing the learned policy of DIO, we found that our agent succeeds in 90% of the time to reach the goal without colliding taking an average of 50 ± 21 steps. Consider that the optimal number of steps without obstacles is 11. It also never exhausts its maximum number of steps, meaning that our agent either reaches a goal or collide with an obstacle in 10% of the time. The policy learned without DIO did not reach a suitable solution and collides with an obstacle 18% of the episodes. Furthermore, it takes an average of 231 ± 59 steps to end an episode, precisely, in 82% of the episodes, the policy exhausts its maximum number of steps and never reaches the goal.

The dip in the mean frames in 13 show that the agent did not only learn how to only collide but also is learning how to reach the goal in the shortest amount of time. Since 11 is still increasing, we can also tell that the values did not converge yet and the agent can learn even a better optimization while 12 shows that the values converge towards a suboptimal policy.

In 11 and 12, it is evident that the policy learned by DIO is more optimal. Similarly, when evaluating our policies on the same reward scheme, the implementation with DIO averages a 0.64 while without DIO averages a -0.18 return.

Overall, the results show that an implementation that makes use of DIO is beneficial even in a stochastic environment. As we lose in speed, we gain in optimality of the solution. Given those preliminary results, we want to (1) optimize our implementation and (2) test our implementation in a more complex scenario that requires a more complex understanding of the world from DIO side. (1) is discussed in 6 and (2) is discussed in 7.

Observation We tested our implementation on a 16x16 gridworld to see how it behaves in an entirely new environment. The policy succeeded in 94% of the runs to reach the goal, while the implementation without dio, never reached it. This is left for later investigation.

6 Optimization

6.1 Compile once Evaluate many

Since the grounding of the problog program is dependent on the query, it is possible to first compile the model to a problog internal format and ground all possible queries and evidence specified in the model. This allows us to compile once and evaluate many. This can help with the optimization of our program, since our files can be defined into (1) rules, (2) world knowledge base and (3) labels as explained in 4.2. (1) and (3) are constants, while the dynamics of the world (2) change at every given timestep. We can then initially ground the constants and later extend the database to incorporate the environment knowledge.

Further work needs to be done as the grounding breaks the multiprocessing capabilities of training the reinforcement learning. Another direction could be to incorporate evidence-based probabilities which allows us to query DIO easily.

6.2 Feedback n-steps

Another possible optimization is to query DIO every n steps, rather than every 1 step. In the following, we evaluate the tradeoff between optimality and speed of training when increasing n.

n-Steps	μ	σ	*min*	*max*
1	49.6	21.0	15.0	139.0
2	52.4	21.5	8.0	124.0
3	180.4	88.0	3.0	256.0
4	232.2	60.2	23.0	256.0

Table 1: n-Steps Frames/Episode

n-Steps	μ	σ	*min*	*max*
1	0.64	0.55	-1.0	0.95
2	0.57	0.61	-1.0	0.94
3	-0.50	0.50	-1.0	0.00
4	-0.15	0.36	-1.0	0.45

Table 2: n-Steps Returns/Episode

As expected by decreasing the number of times we call DIO, we were able to reduce the training time considerably. By half for $n = 2$, a third in $n = 3$, etc. Overall, the change of optimality between $n = 1$ and $n = 2$ is negligeable, as the success percentage drops from 90% to 87%. The mean return drops by 0.9, although the max reward does not change much. Starting from $n = 3$ and $n = 4$, we gain a lot in performance but we lose almost all optimality. A visualization of the resulting policy shows that the agent almost never reaches the goal, similarly to the implementation without DIO. In conclusion, $n = 2$ is the best optimization.

7 Traffic Simulation

Given the definition of our DIO architecture, the performance as shown in Figure 2 will be judged on a Traffic Simulation. Precisely, we make use of the OpenSource code of Carla [19] to put our implementation in practice. This is a *work in progress*.

7.1 Scenario in Reinforcement Learning

Our autonomous agent is one vehicle on the road. The road is populated with other vehicles, walkers and traffic signs and lights. The goal of the agent is to get from point A to point B on the map in the fastest time possible. Carla defines *sensors* to collect data from the world. From the sensors, we collect **collision detector**, **obstacle detector**, **IMU** (which defines the internal state of the vehicle that includes acceleration) and finally **position** data to incorporate into our state representation. Thus,

$$S_t = [t, p, c, o, a]$$

In other words, the state at time t is defined by t the time, p the position and a the acceleration. c and o are both flags (-1/1) that indicate respectively whether there is a collision or an obstacle detected. (-1) is equivalent to detected while (1) is equivalent to non-detected. Furthermore,

$$A_t =](a_t) - min, (a_t) + max[$$

Precisely, our action space at time t is a range delimited by the minimum acceleration possible and the maximum acceleration possible. Finally,

$$R_t(s_t) = c * (d(i, p)/t) + (o * a)$$

We define d a function that takes in the initial position i and the current position p and computes the distance travelled. By putting everything together, we can see that the reward is maximized when the agent travels the longest distance in a shortest amount of time (without colliding at high speed) while making sure that when an obstacle is detected, acceleration is kept low.

7.2 Domain Specific Rules in DIO

Note that DIO is a logic programming module that we design in Prolog and carry to Python via PySwip [20]. The first step of DIO is to consider the translation.

$$\frac{\text{Time}(t) \wedge \text{Pos}(p) \wedge \text{Acc}(a) \wedge \text{Collision}}{[t, p, -1, 1, a]}\ T_1 \qquad \frac{\text{Time}(t) \wedge \text{Pos}(p) \wedge \text{Acc}(a) \wedge \text{Collision} \wedge \text{Obstacle}}{[t, p, -1, -1, a]}\ T_2$$

$$\frac{\text{Time}(t) \wedge \text{Pos}(p) \wedge \text{Acc}(a)}{[t, p, 1, 1, a]}\ T_3 \qquad \frac{\text{Time}(t) \wedge \text{Pos}(p) \wedge \text{Acc}(a) \wedge \text{Obstacle}}{[t, p, 1, -1, a]}\ T_4$$

The domain specific rules will follow the same idea, taking in the ground facts and the action, and infering the next set of ground facts. Thus, DIO searchs for configurations of ground facts given the step semantics, to return the set of labels. The main challenge is pruning the search space as we want to guarantee time efficiency. While we hypothesize that the RL would learn faster by incorporating domain knowledge, we are expecting iterations to take longer as the feedback loop incorporates DIO computation. This part is an ongoing work.

8 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 [21, 22, 23, 24]. [24, 22] 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 [24] 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, [22] 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 [21]. Precisely, in [21], 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 [22, 21] 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.

9 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. This is a task we were able to produce in our simpler introductary example of the Dynamic Obstacles in the grid world. Results are promising and will be further extended to the traffic simulation referred to in 7. More directions open up as we think of optimization, this includes mix-matching the n-steps approach with the number of steps DIO can look ahead. Similarly, we look at the differences between overwriting vs. fine-tuning the rewards using DIO and if such choice matters in training. As we start adding complexity to the algorithm, we turn our focus into the specifications as shown in A.3 to better inform on meaningful and effective way to translate our labels to their corresponding numerical values design choice.

9.1 Next Steps

Our results of training with the Reinforcement Learning alone are suboptimal. We are investigating why that is the case by playing around with the reward function from the RL side. So far, we have tried punishing when close to an obstacle and reward when close to the goal. Changing the values has unfortunately not changed the policy and we are still looking into more optimal results for the RL side. While we answer those questions, we will focus on exporting the architecture to a more complex scenario: *Carla*. This will give us a better indication on the success of DIO. Note that multiple directions have been mentioned in the following thesis as future investigations. A breakdown of those will be briefly given here. (1) How does overwriting vs. fine-tuning the previous reward affect the policy? (2) How does the policy with DIO behave in a completely new environment? (3) How does adapting the steps DIO look ahead with the n-steps feedback optimization affect the policy?

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

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

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

[18] Samar Rahmouni and Giselle Reis. Domain informed oracle for reinforcement learning. `https://github.com/natvern/Thesis`, 2022.

[19] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[20] Yüce Tekol. Pyswip. `https://github.com/yuce/pyswip`, 2020.

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

[22] 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.

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

[24] 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 15.

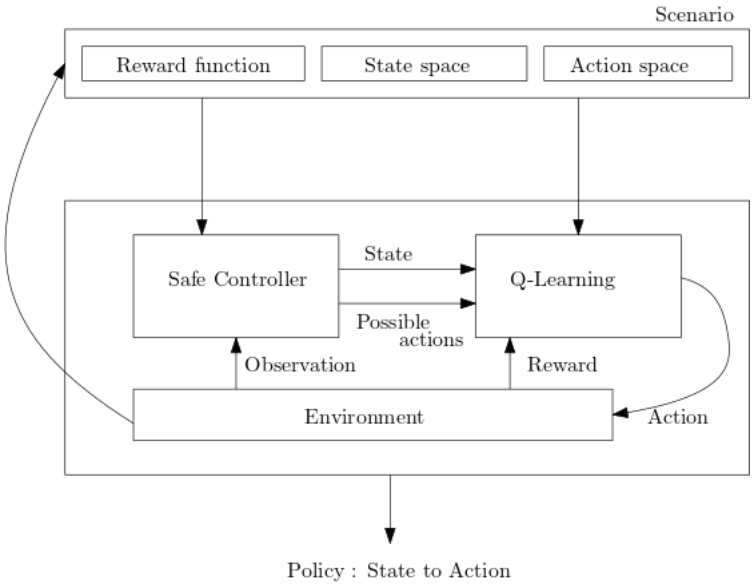

Figure 15: Overview of the proposed solution

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

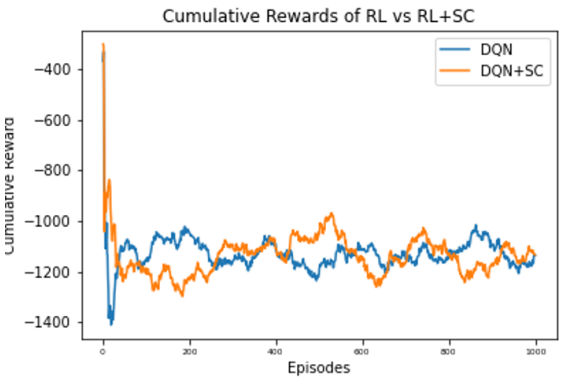

Figure 16: Cumulative Rewards of RL vs. RL+SC

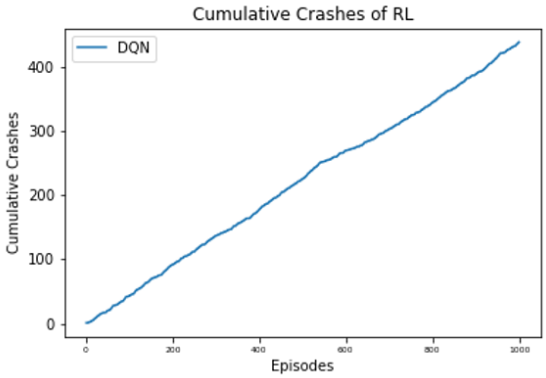

Figure 17: Cumulative Crashes of RL during Training

The graph in Figure 16 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 17 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.

A.2 Reinforcement Learning: Behind the Scenes

Reinforcement Learning is based on getting sensory input, making a decision, and getting either rewarded or punished for it. This method of learning is made possible and effective considering the environment is a *Markov Decision Process*.

Markov Decision Process (MDP) Mathematical framework for environments where the outcome of a decision are both affected by its stochasticity (e.g. failure in motors) and the decision making of the agent. Consider this simple scenario, there exists an agent in a maze whose goal is to escape. The agent at some time t observes that there exists walls to his left and to its right, call this state s_t, he might then

decide to go up, this can either result with probability p into the action actually executing as expected, thus moving him a square up to state s_{t+1} or it could fail with probability $1-p$ into going right instead, thus resulting in state s'_{t+1}. This is Markov Decision Process. Note specifically that the outcome only depends on what happened a state before. This is similar to 19 where worlds are instead referred to as states. This consideration allows us to define a reward function $R_t(s,s')$ given action a. The agent goal is then to maximize its returns and come up with a policy that given a state returns the action to take. Consequently, we are able to define a value V to a given policy by computing the expected reward from following it. Precisely,

$$V_\pi = E[R \mid s_o = s]$$

There are multiple ways to think about how to maximize this function, those result in different algorithms for Reinforcement Learning. In the following section, we present those we've been using.

Q-Learning Q-learning considers the potential reward as the weighted sum of the expected values. It considers the following hyperparameters. (1) α the learning rate associated with how much we care about new learning vs. old one, (2) γ the discount factor associated with the importance of future rewards. At the core of Q-learning is the *Bellman Equation* which results in the following equation.

$$Q_{t'}(s_t,a_t) = Q_t(s_t,a_t) + \alpha \times (r_t + \gamma \times max_a Q(s_{t+1},a) - Q(s_t,a_t)$$

Precisely, $max_a Q(s_{t+1},a)$ considers the maximum possible future reward. At its core, the Q-value of a given (state, action) pair is dependent on (1) the current value given the learning rate, (2) the reward if we are to take a and finally (3) the maximum expected reward that can be obtained in the resulting state.

Proximal Policy Optimization (PPO) We mentioned previous how most of the effort in Reinforcement Learning is fine-tuning the hyperparameters and coming up with an effective reward function. PPO strives to find a balance between this fine-tuning and ease of implementation. It ensures so by reducing the deviation between π_t and π_{t+1}.

A.3 DIO interface

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 18. Note that DIO follows a certain interface which we describe in the procedure below.

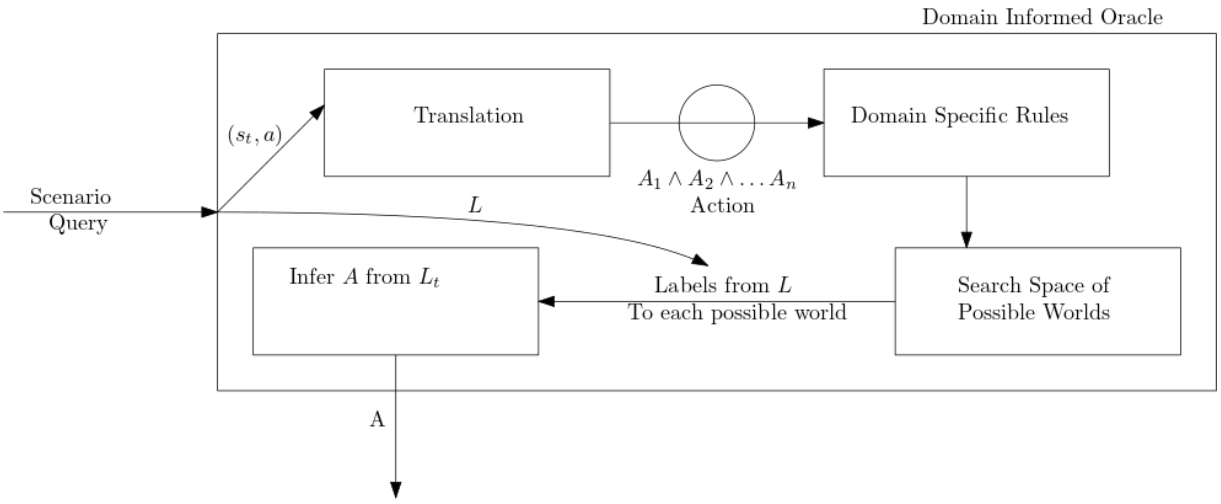

Figure 18: Domain Informed Oracle routine

1. The scenario queries DIO. It sends (s_t, a), the state at time t and the action. It also sends $L : s \to t$ 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 $T : \mathbb{R}^n \to o_1 \times o_2 \times ...o_n$ 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 no certainty on whether the worlds expected by DIO will necessarily happen. Consider Figure 19.

5. Those possible worlds, equivalent to states, can be transformed using L to t. This is done in a declarative tool, problog in our case. In practice, t this labeling takes in the ground facts and generates possible predicates.

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 J judgment from S is formulated. For instance, we can consider a rule where if most of the possible worlds are undesirable, then let J be undesirable. This judgment can only be a numerical value as we've done in our gridworld example. Precisely, J is the sum of the weighted probabilities of our system at the next timestep.

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

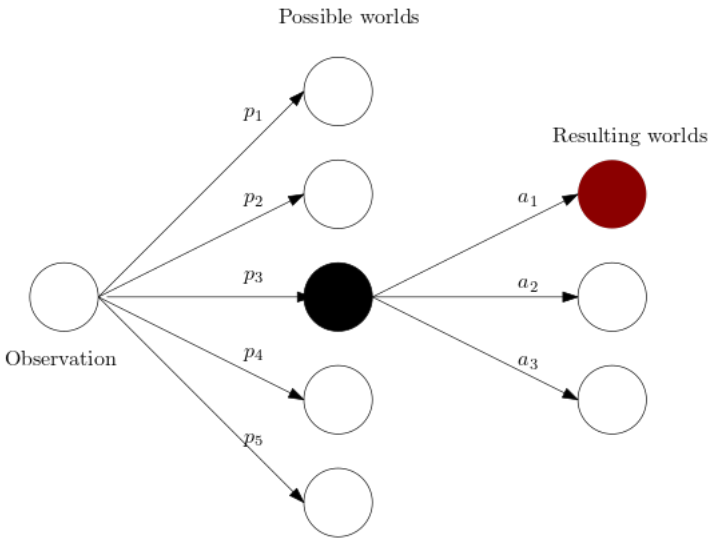

Figure 19: Game Tree Simulation by DIO

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. However, note that those problems are existing problems in defining a reward function without DIO but made easier to investigate with DIO. This is made possible because the problem is not that of writing the reward function given a state vector, rather how to write a reward function given A. For instance, in our experiments, we label states as either desirable, undesirable or fatal and associate the corresponding numerical values $(1, -0.5, -1)$, intuitively. Such a basic labeling gave us good results without going through the trial-and-error process that is present in RL.

Terminology

discount factor Hyperparameter defininining how much future rewards affect the current decision (i.e. a high discount factor means we care less about the future). 22

episode Collection of steps before the goal is achieved OR the agent terminates (i.e. crash into obstacles). 9, 22

exploration vs. exploitation Dilemma in Reinforcement Learning that defines whether to choose an unexplored or explored state at a given timestep. 22

frame Step taken by the agent that advances the environment. 9, 12, 22

learning rate Hyperparameter defining how slowly or quickly the agent learns from experience. 22

meta-learning How well an agent trained in one environment adapts to a new environment. 22

norm Textual guides translated into the reward function. 10, 22

observation The input gathered by external or/and internal sensors. 22

policy Resulting function of training that maps a state to an action. 12, 22

reinforcement learning (RL) Trial-and-error AI algorithm based on Markov Decision Process and the Bayesian rule. 22

return Summation of rewards for a given episode. 9, 12, 22

reward Numerical value often normalized given to the agent after execution of an action. 22

reward shaping Concept behind adapting the reward function to minimize human bias. 22

state Internal representation of the environment specific to Reinforcement Learning. 22