Domain Informed Oracle for Reinforcement Learning

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

Reinforcement learning (RL) is a powerful AI ~~method~~technique that does not require pre-gathered data but relies on a trial-and-error process for the agent to learn. This is made possible through a reward function that associates ~~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, since interpreting abstract states in RL in the context of a domain needs to be done on a case by case basis. ~~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 vectors 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~~to systematically incorporat~~ing~~e 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.~~states and rules can be defined conveniently. For each state and action pair, DIO provides information ~~an informed decision~~ to the reward function, to dynamically adapt itself ~~thus allowing it to dynamically adapt the rewards~~.

Our implementation is tested on a grid world with dynamic obstacles. ~~and later extended to a traffic simulation scenario. It is then~~ 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, the success percentage, i.e. the number of times it reaches a positive terminal state, a goal for example.~~ The results show that the domain specific knowledge incorporated easily through DIO lead to shorter training times, faster convergence, and fewer errors.

~~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.~~

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

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

ACM Reference Format:
Samar Rahmouni and Giselle Reis. 2022. Domain Informed Oracle for Reinforcement Learning. 1, 1 (November 2022), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

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

G: Abstract should be shorter. I crossed out what could be removed to achieve this.

G: This can be elaborated in the introduction.

G: This can be elaborated in the experiments section.

G: Speculation... Rewrite once results are in.

G: I think this is not updated...

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 [15], 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 MOTIVATION

Reinforcement Learning is a method of learning that maps situations to actions in order to maximize its rewards [19]. 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.

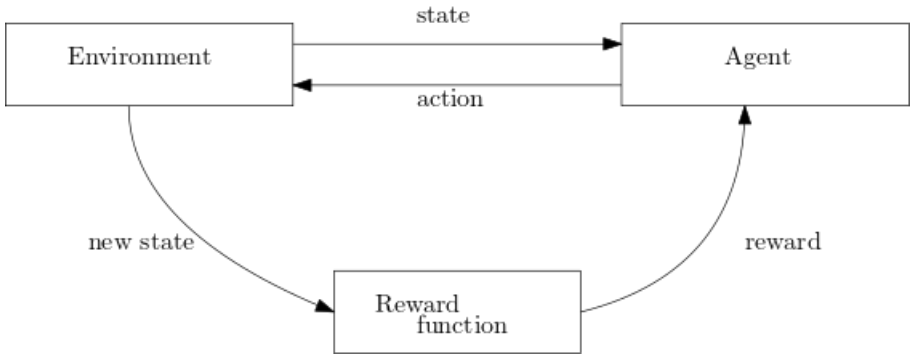

Fig. 1. Reinforcement Learning Routine

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

The reward function is a crucial aspect of the RL algorithm. For instance, consider a game of chess where the agent is punished when it loses and rewarded if it wins. The agent is bound to learn how to maximize its winnings but it will need to exhaust multiple possible combinations to learn. In this case, the training time is not optimal. A better approach would be to also reward it for making a good opening, for instance. Another example would be only considering negative rewards. Say we want our agent to escape a maze, and we punish it at every timestep for not escaping. If there is a fatality state (*e.g.*, a fire or a black whole), the agent will learn to move towards the fatality state as to cut its negative rewards as soon as possible. In conclusion, a good reward function is the first step of optimal learning. By choosing 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 **??**.

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 [14] 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 [13]. 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 [20]. 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 [11]. 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 [10, 18]. 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).

2.2 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 [4], 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 [2]. As a result, neurosymbolic AI has been successfully applied to vision-based tasks such as semantic labeling [12, 21], vision analogy-making [16], or learning communication protocols [8]. Those results inspire us to use those techniques for reinforcement learning, as to tackle its challenges.

3 DOMAIN INFORMED ORACLE

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

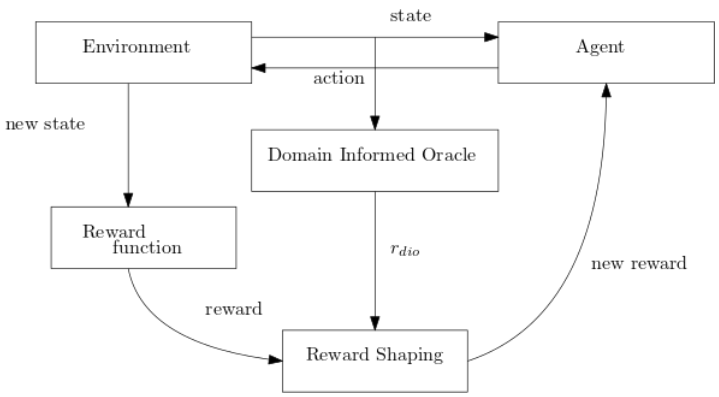

Fig. 2. Overview of the proposed solution

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

We could leave the *reward shaping* unit as a design choice. However, it will be enforced in add section where reward computation that whether it is left as a design choice or else, is irrelevant given howDIO affects the final results.

3.1 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 ADD REFERENCE TO PPO PAPER. It does not mean that our solution is specific to it, rather it can be generalized to any algorithm choice.

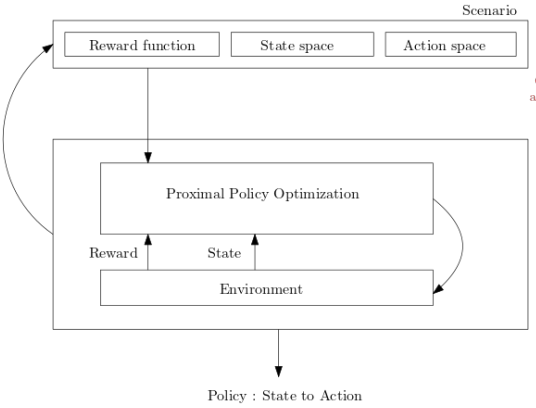

Fig. 3. Reinforcement learning architecture

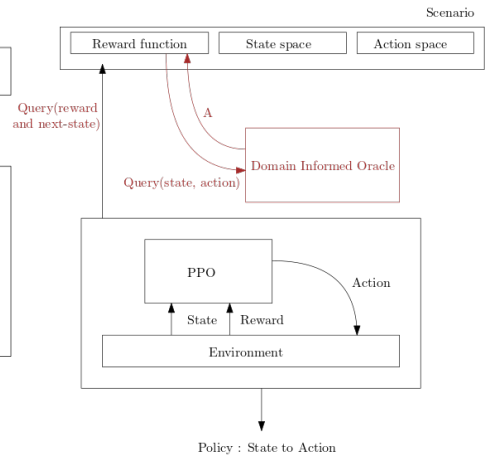

Fig. 4. DIO+RL architecture

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

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

3.2 DIO procedure

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

(1) **World Rules** defining the rules governing the world. This is domain-dependent and implemented in a logic programming file, i.e. we are able to define the next step via step semantics.
(2) **Knowledge Base** defining the ground facts which describe the world at a given time step. This module is continuously updated to account for the dynamics of the state.
(3) **Labels** i.e., textual "norms" corresponding to an iteration of the state. In practice, they are all possible judgments on the resulting state, e.g. *crash :- obs(X,Y), agent(X,Y),*

or *maybecrash :- nextObs(X,Y), agent(X,Y).* Those labels have probabilities associated with them and indicators. Precisely, an indicator over a negative state is -1.

(4) **Translation Unit** defining the translation from state to ground facts and from labels to a numerical value, e.g. if the predicate crash is true with $P = 0.25$, then the reward shaped is $r + -0.25$.

(5) **Reinforcement Learning** is our independent module that does not make assumption on the algorithm chosen for RL.

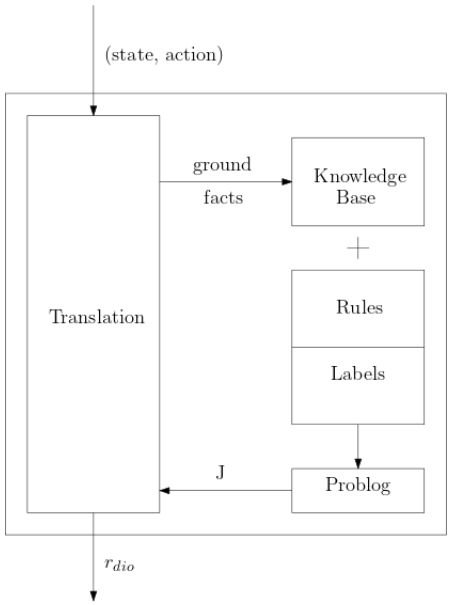

Fig. 5. Modules & Interactions

Figure 5 describes the interactions of the different modules. Precisely, given the rules and the world knowledge base at a given time t, we are able to produce the corresponding label, i.e. the query over a predicate. The predicate is fed into the Translation Unit (TU) that transforms the predicate to a numerical value that is given to the Reinforcement Learning as a reward shaping r'. . Note that the inference on the query is done by a declarative tool that incorporates probabilities called *Problog* that we introduce in 3.3.

3.3 Problog Procedure

Problog is a logic programming language that aims to bridge between probabilistic logic programming and statistical relational learning [7]. 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.

3.4 Computation of Final Reward

In general, reward shaping is expressed as follows $R = r_{rl} + r'$, such that r_{rl} is the original reward defined by the reinforcement learning environment. Usually, this associates the value of 1 for a positive terminal state, and -1 for a negative terminal state. r' is the human bias, i.e. what is added to "shape" the reward function. Although only rewarding terminal states is enough to 'reach' a policy that will minimize its losses, it is important to note that a reward function should consider the cost of a 'step' for optimal and fast training. The cost of a step is dependent on the state given at that step. For instance, an autonomous vehicle stopping with not cars around, is different from a vehicle stopping while a car is behind it. Although the states are not 'terminal', i.e. they do not end an episode, they have different weights. An optimal reward function is one that considers those differences. A logic programming module allows us to infer over states using a knowledge base and world rules seamlessly.

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

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

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

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

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

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

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

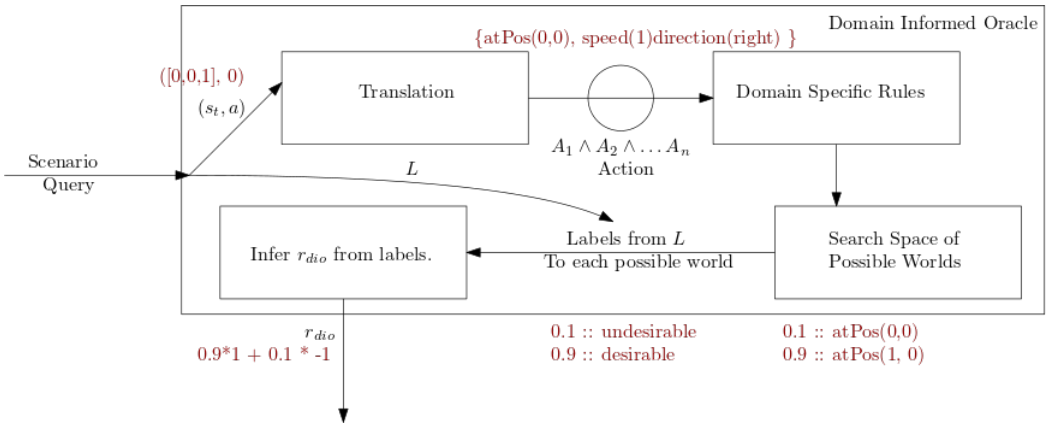

Fig. 6. Step-by-step of Computed Final Reward

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

4 DYNAMIC OBSTACLES IN A GRIDWORLD : WORKING EXAMPLE

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

4.1 Scenario in Reinforcement Learning

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

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

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

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

Finally, the reward is a straightforward one that reward 1 for reaching the goal, -1 for failing to do so, either by colliding with an obstacle, or exhausting its battery (maximum number of steps). Furthermore, for every step the agent takes, it has to spend the 'cost' of living, which is $-\frac{1}{n}$, where n is the size of board.

4.2 Domain Specific Rules

The rules are defined as a ProbLog [5]: a probabilistic prolog that allows us to capture the stochasticity of the environment, as we previously introduced in 3.3. Precisely, we want to consider the erratic movements of the obstacles, considering we do not have previous knowledge on the distribution of their given movement. We assume a uniform distribution and define the following. The rules of DIO deriving a predicate γ take the following form:

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

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

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

(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 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.

4.3 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,

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

4.4 From Norms to Labels

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

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

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

4.5 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, r_rl = Step'(S_t, ac)                          ▷ Call to Initial Env
8:      r'_DIO = getFeedback()                              ▷ Query DIO
9:      R = TU.getReward(r_rl, r'_DIO)
10:     return (obs, R)
```

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

Algorithm 2 Inference of Judgment r'_{DIO}

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

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

5 METHODOLOGY

5.1 Metrics

We compare our architecture incorporating DIO to a reinforcement learning architecture. Precisely for our scenario presented in SECTION, we judge *optimality*, i.e. the number of steps to reach the goal and *safety*, i.e. ratio of successes to failures. Those metrics are analyzed during both training and deployement. We gather the cumulative rewards and cumulative rewards as well as the total number of failures and success over 180k frames of training, and 1000 episodes of deployement.

5.2 Settings

Our above metrics and gathered for 8 different settings. We consider both α, 0 (equivalent to rl alone), 0.5, 1, 3 and 9. Similarly, we consider three settings for the number of obstacles: (1) minimal (1/20 of the board is covered by obstacles), (2) intermediate (1/10th) and finally, (3) extreme (1/3rd). For each, we gather the metrics described above.

5.3 Gathered Results

During Training. For optimization purposes, training is done using multiprocessing reinforcement learning as indicated in figure 7. The network is loaded and parallel environments are spawned

given the available number of processes. For n frames, the rl routine is returned before the final experience is collected and used to update the parameters of the neural network. For our purpose, the cumulative reward is the reward of each (obs_i, r_i) added over the number of updates. Similarly, the terminal states are computed according to this similar philosophy, meaning that at every given environment, if the resulting state is terminal and a failure, we add it to the cumulative failures to assess safety overtime. At the end of training, we compute the ratio of successes over failures to analyze the needed number of failures before convergence.

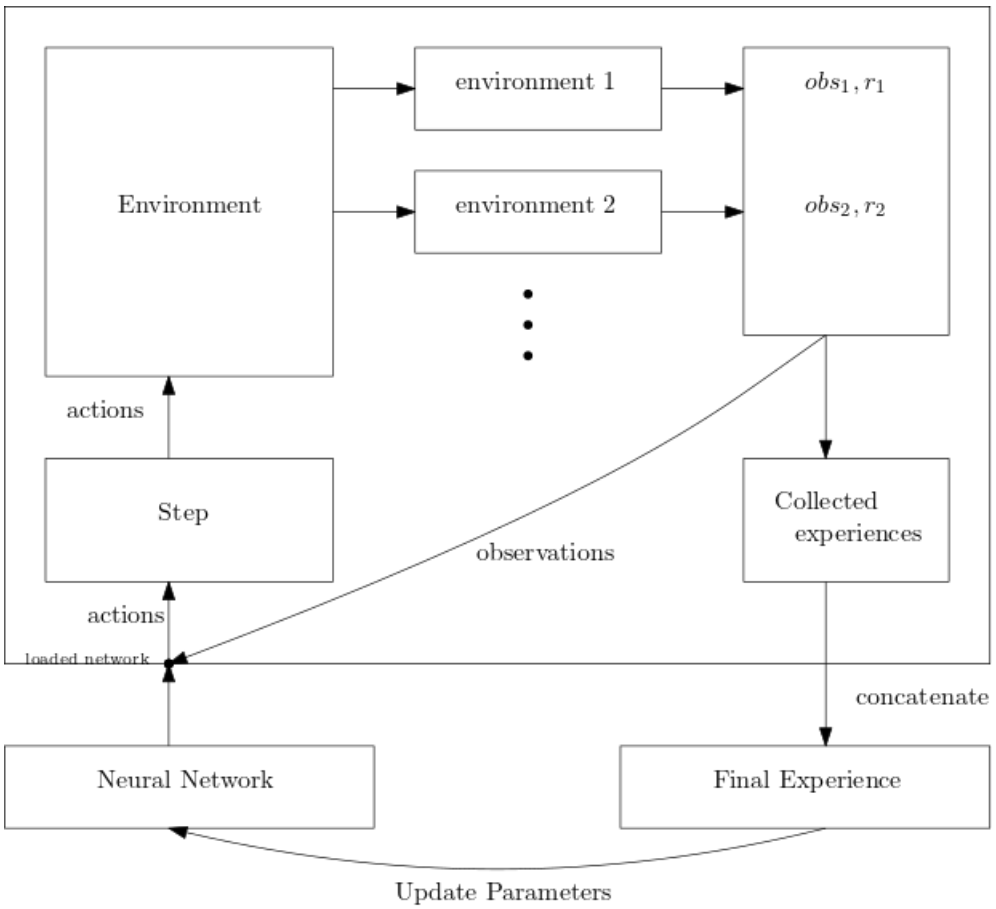

Fig. 7. Multi-processing Reinforcement Learning

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

6 RESULTS

The FINAL PART AAAA.

7 RELATED WORK

As discussed previously in Section 2.2, neurosymbolic AI trends show promising results in improving ML algorithms, whether that is from an interpretability aspect or an optimization one. More recent works take this trend and incorporate symbolic reasoning and domain knowledge in reinforcement learning settings [1, 6, 9, 17]. [9, 17] 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 [9] 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, [17] 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 [6]. Precisely, in [6], 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 [6, 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.

8 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 **??**. 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 **??** to better inform on meaningful and effective way to translate our labels to their corresponding numerical values design choice.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

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

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

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

[4] Artur d'Avila Garcez and Luis C. Lamb. 2020. Neurosymbolic AI: The 3rd Wave. arXiv:2012.05876 [cs.AI]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009