Developing a Python Reinforcement Learning Library for Traffic Simulation

Gabriel de O. Ramos  
Instituto de Informática  
Universidade Federal do Rio Grande do Sul  
Porto Alegre, RS, Brazil  
goramos@inf.ufrgs.br

Liza Lunardi Lemos  
Instituto de Informática  
Universidade Federal do Rio Grande do Sul  
Porto Alegre, RS, Brazil  
illemos@inf.ufrgs.br

Ana L. C. Bazzan  
Instituto de Informática  
Universidade Federal do Rio Grande do Sul  
Porto Alegre, RS, Brazil  
bazzan@inf.ufrgs.br

ABSTRACT

Evaluating multiagent reinforcement learning (MARL) approaches in real world problems, such as traffic, is a challenging task. In general, such approaches cannot be deployed before extensive validation. Hence, simulating the impact of these approaches represents an essential step towards its deployment. Existing MARL tools make this process easier by simulating real world scenarios. However, no such framework simulates traffic with the required level of detail. Alternatively, one may consider hand-coding such simulations. Nonetheless, this adds another complexity layer to the process. In this paper, we introduce PyRL, a Python Reinforcement Learning Library that facilitates the development and validation of MARL techniques. PyRL implements well-known RL algorithms and validation scenarios. Additionally, in contrast to existing tools, PyRL delivers detailed traffic scenarios through the SUMO traffic simulator. Moreover, PyRL can be more easily extended than existing tools. In short, PyRL simplifies the validation of MARL approaches (especially in traffic scenarios), requiring little programming efforts and speeding up the setup of experiments.

CCS Concepts

• Computing methodologies → Modeling and simulation; Multi-agent reinforcement learning; • Applied computing → Transportation; • Software and its engineering → Development frameworks and environments;

Keywords

reinforcement learning, multiagent, Python, library, traffic, simulation

1. INTRODUCTION

The development and evaluation of multiagent reinforcement learning (MARL) techniques in real world problems is far from trivial. Such a task involves simulating an environment’s dynamics as well as the agents’ behaviour and interactions [19, 20]. Nevertheless, MARL has been successfully applied to (and delivered promising results in) several domains, such as traffic [3], smart grids [17], robotics [8], among others [13].

A particularly relevant domain is that of traffic. It is well known that traffic issues are faced everyday even in small cities. As such, traffic has drawn special attention from the Artificial Intelligence community. In particular, given its distributed and selfish nature, traffic has shown an interesting testbed for MARL algorithms. However, an important aspect to consider here refers to the way such scenarios are validated. In general, the deployment of new technologies in traffic domains is only attainable after extensive experimentation. Thus, traffic simulation emerges as a safe and economically efficient way of validating such scenarios. Here, one needs to model drivers’ behaviour and to simulate vehicles within the road network. A representative tool here is the SUMO simulator1, which models the system at a microscopic level, i.e., even the vehicles’ position and speed are simulated. We refer the reader to Bazzan and Klugl [4] for a more detailed overview of agents in traffic and its simulation.

Regardless of the problem being addressed (and of the way it is simulated), however, the most important aspect here refers to putting the RL algorithms into the agents. On the one hand, the most straightforward approach is to hand-code the algorithm and the environment (or an abstract representation of the environment). However, such requirements make the validation process inefficient and slow. Additionally, code reuse is not always attainable. On the other hand, existing libraries make this process easier by providing algorithms and validation scenarios. Notwithstanding, no such library models traffic with the required level of detail. Furthermore, extending such libraries to work with realistic traffic environments (e.g., using the SUMO simulator) is not a simple process.

In this paper, we go beyond existing RL libraries and introduce PyRL – a Python Reinforcement Learning Library2. PyRL is focused on facilitating the development and validation of (MA)RL techniques. Specifically, PyRL includes implementations of well-known RL algorithms (e.g., Q-learning, WPL) and validation scenarios (e.g., normal form games, cliff walking). The validation scenarios (or, as called, environments) are modelled as Markov Decision Processes (MDP) [19], which are compatible with the implemented RL algorithms. In contrast to other libraries, PyRL also includes traffic environments (simulated using SUMO), which allows for the validation of MARL algorithms in highly detailed traffic simulations. In short, PyRL enables code reuse, has little programming efforts and speeds up the setup of experiments.

We remark that PyRL is still under development. In spite of that, PyRL is a powerful tool for developing and validating MARL approaches, especially in traffic scenarios. PyRL

1https://www.sumo.dlr.de/
2https://github.com/goramos/pyrl
is written in Python and has SciPy as its main dependence. As for the traffic environments, PyRL also requires the SUMO simulator and the `py_expression_eval` library.

This paper is structured as follows. Alternatives to PyRL are presented in Section 2. Section 3 details PyRL’s architecture. Examples on how to use and extend PyRL are discussed in Sections 4 and 5, respectively. Final remarks are presented in Section 6.

2. RELATED TOOLS

In this section we review some of the alternatives to PyRL and discuss their drawbacks. Recall that the objective of PyRL is to enable simpler and faster validation of (multiagent) RL algorithms in different scenarios, especially in traffic. The most straightforward alternative to PyRL refers to hand-coding algorithms and experiments from scratch. However, this is a time-consuming task, which requires extensive verification and is subject to failures. Moreover, this approach neglects code reusability.

Existing RL frameworks represent another alternative to PyRL. An interesting one is the Reinforcement-Learning-Toolkit [18], which comprises a collection of RL algorithms and demos. However, algorithms and scenarios are not independent modules, but part of a single code. Consequently, changing an algorithm or scenario may be complex. In fact, such a toolkit would be better defined as a set of RL examples than an RL library itself.

Another relevant tool is OpenAI Gym [6], which provides several environments for testing an RL algorithm. The focus of OpenAI Gym is on testing and comparing RL algorithms and making the results publicly available. As such, OpenAI Gym only provides the environments, not the algorithms. The lack of algorithms is the main limitation of OpenAI Gym as compared to PyRL. In fact, implementing state-of-the-art algorithms is frequently far from trivial. Additionally, no traffic scenario is provided with OpenAI Gym.

A final worth mentioning tool is PyBrain [16], a machine learning library that also includes RL algorithms and problems. In terms of functionality, PyBrain is similar to PyRL. Both PyBrain and PyRL can be extended with additional modules, though PyBrain includes more algorithms and scenarios natively. However, PyBrain’s code is overly modularised, making the inclusion of new algorithms and environments more difficult than in PyRL.

As seen, existing tools and libraries do not achieve the same level of extensibility as PyRL does. Moreover, PyRL is the only RL library with native integration with a traffic simulation tool (i.e., SUMO simulator). Although the other platforms can be extended with traffic scenarios, this is not a trivial task and may involve extensive validation. Hence, PyRL represents one of the easiest and fastest ways of validating RL approaches in traffic domains.

3. ARCHITECTURE

PyRL simulates a reinforcement learning problem through abstractions of its environment and agents. The basic structure of PyRL is depicted in Figure 1, where Environment models the problem under consideration and Learner models a learning agent that interacts with the environment. PyRL provides a few different environments and learning algorithms. The environment is represented as an MDP, which can be simulated internally or externally (e.g., using a third-party simulator). For instance, the simulation of traffic scenarios in PyRL is performed using the SUMO simulator. An environment supports one or more agents. Each agent is an instance of the Learner class, which implements a reinforcement learning algorithm. The ExplorationStrategy class provides exploration strategies to be used by the learning algorithms. More details on each of these classes are presented in the next subsections.

The basic steps for setting up a PyRL simulation are as follows. Firstly, an environment must be instantiated and its parameters defined. Secondly, the agents must be instantiated together with its parameters and exploration strategy. Finally, the simulation can be run for any given number of episodes. The main advantage of PyRL over hand-coding an experiment from scratch (e.g., implementing an environment and a learning algorithm) is that once an Environment and a Learner are created, they can be used with any other available PyRL module with little effort. Examples on how to extend PyRL are presented in Section 5.

3.1 Environment Class

The Environment class models a reinforcement learning problem by means of an MDP. Specifically, it provides the abstract implementation of an MDP and the methods required for controlling and interacting with it. Observe that Environment only provides the abstract structure required by all PyRL environments. An environment itself and the corresponding MDP are implemented through a subclass of Environment. In other words, the states, actions, transitions and rewards of the MDP are only implemented in the child class. In order to create an environment, the following abstract methods must be implemented:

- `__init__(self)`: initialises the attributes of the environment, including a data structure for representing the MDP.
- `get_state_actions(self, state)`: returns the set of actions available in the specified state.
- `run_episode(self, max_steps)`: runs a complete simulation episode (or up to max_steps). The episode duration (number of steps) depends on the problem being modelled.
- `run_step(self)`: runs a single step within an episode. In a step, all agents choose their actions and receive the corresponding rewards.
Whenever an environment is instantiated, its __init__ method initialises its attributes and creates the internal representation of the MDP. The same is done with the agents (as shown next, in Section 3.2). The Environment super-class keeps a list with the agents working on it. This list is updated with the register_learner method, which is called by each agent once created. The get_state_actions method enables the agents to observe the set of actions available in a given state. This method is specially useful when the agents are exploring unknown states.

The run_episode method is responsible for controlling a complete episode of the simulation. As such, it runs an step for the specified number of times, or up to a point where all agents have reached their goals. The run_step method is more specific, and implements the interaction of the agents with the environment on each simulation step. Precisely, it processes the agents’ actions, computes the current state of the world, and then provides the corresponding reward to the agents. This is the basic flow of an RL problem.

In the current version of PyRL, the following environments are available:

- **CliffWalking**: a single-agent, 4x12 grid-world environment [19]. The agent starts in the lower left corner (starting state) and must cross the environment to reach the lower right corner (goal state). However, there is a cliff between the starting and goal states, which should be avoided by the agent. The MDP is modelled as follows. Each grid cell represents a state. In each state, the agent can move in four directions: right, left, up, and down (some movements have no effect in edge states). Each visited state incurs a reward of -1, except for the cliff and goal states (a.k.a. ending states), which incur a reward of -100 and +100, respectively. Transitions are deterministic, i.e., the action taken is always the actually chosen one.

- **TwoPlayerTwoAction**: a generalisation of two-player-two-action normal form games [9]. This class provides some classic games (such as prisoners dilemma, matching pennies, coordination game, battle of the sexes) and allow the creation of user-defined ones as well. The game to be used must be specified at the time of instantiation.

- **SUMO**: a multiagent traffic scenario simulated using the SUMO simulator. In this scenario, each agent has an origin and a destination, and must find a route that minimises its travel cost. The road network follows the SUMO specification and must be specified at the time of instantiation. The MDP is modelled as follows. States are intersections. The actions in a particular intersection correspond to its outgoing roads. Transitions are deterministic. The reward for crossing a road is its negative travel time (i.e., a travel time of 10 is interpreted as -10 reward). Observe that the agents do not know their routes a priori here, i.e., they must learn their route link-by-link.

- **SUMORouteChoice**: another multiagent traffic scenario simulated with SUMO. As opposed to the SUMO environment, here the agents know their routes a priori. As for the SUMO environment, the road network follows the simulator specification and must be specified at the time of instantiation. This scenario is modelled as a stateless MDP. The actions of an agent correspond to the available routes between its origin and destination. Again, transitions are deterministic and the reward for crossing a route corresponds to its negative travel time.

### 3.2 Learner Class

The Learner class models an agent by means of a reinforcement learning algorithm. Basically, it provides abstract methods for acting in the environment and receiving feedback from it. As for the Environment class, Learner only provides the abstract structure required by all PyRL learners. Any RL agent is then modelled as a subclass of Learner. The following abstract methods are provided with the Learner class:

- **__init__(self)**: initialises the learner’s attributes, including those of the learning algorithm (such as the Q-table, in the case of Q-learning).

- **act(self, state, available_actions)**: chooses and returns the action to be taken by the agent in its current state. The way such decision is made depends on the algorithm being implemented and may employ an ExplorationStrategy. The current state and corresponding actions may be received as parameters, if necessary. This method is divided into five parts (act1, act2, act3, act4, and act_last) to ensure agents are synchronised, if needed (it may be necessary when the agents communicate to each other).

- **feedback(self, reward, new_state, prev_state, prev_action)**: receives the reward corresponding to the action chosen in act. Such reward is used to update the agent’s knowledge in accordance with the corresponding RL algorithm. This method is divided into four parts (i.e., feedback1, feedback2, feedback3, and feedback_last) following the same reasoning of the act method.

The __init__ method is called once, when the agent is created. Here, the agent registers to the environment by calling the register_learner method (as seen in Section 3.1). Afterwards, the agent builds its internal representation of the MDP by calling the environment’s get_state_actions method. In general, the agent can only observe the actions on its current state. As such, the agent’s internal representation of the MDP is built progressively, as the agent explores the environment.

The most important methods of a Learner are the act and feedback ones. At every step of an episode, the agent select an action based on its current state and knowledge. The act method is responsible for choosing such an action and returning it to the environment. The action here may be chosen using an ExplorationStrategy (more details in Section 3.3) or an internal procedure. Afterwards, the environment provides a reward to the agents by calling the agents’ feedback methods. These methods use the received reward to update the agents’ experience (e.g., updating the Q-table in the case of Q-learning). Observe that multiple act and feedback methods are available. The idea here is to synchronise the agents at specific decision points while acting or receiving feedback. For instance, algorithms that employ some kind of communication among the agents may require that, before making a final decision, all agents have made an intermediate decision. In this case, if the agents are
not synchronised (e.g., if some of them have not made the intermediate decision), then the decision process may fail. To this regard, PyRL synchronises the agents by ensuring that all agents finish an act/feedback procedure (e.g., act1) before calling the next one (e.g., act2).

In the current version of PyRL, the following Learner classes are provided:

- **QLearner**: uses the traditional Q-learning algorithm [21].
- **WPL**: implements Weighted Policy Learner [1], which is a gradient ascent learning algorithm.
- **OPPORTUNE**: based on the OPPORTUNE algorithm [12], in which agents communicate to coordinate their actions.

### 3.3 ExplorationStrategy Class

The ExplorationStrategy class models an action selection mechanism for balancing exploration and exploitation. Specifically, it provides the choose method, which receives a set of actions and their expected values (e.g., Q-values) and chooses a single action based on specific criteria.

The current version of PyRL delivers the following action selection strategies (we refer the reader to [19] for more details on these strategies):

- **EpsilonGreedy**: implements the $\epsilon$-greedy exploration strategy. Here, a random action is chosen with probability $\epsilon$ and a greedy (with highest Q-value) one with probability $1-\epsilon$.
- **Boltzmann**: implements a soft-max action selection mechanism using the Boltzmann distribution. Here, the probability of an action is proportional to its value. Exploration and exploitation are balanced by means of parameter $\tau$.

### 4. EXAMPLES OF USE

In this section we present two examples on how to setup an experiment in PyRL. We start with the classic cliff walking scenario (Section 4.1) and then present a traffic scenario example (Section 4.2).

#### 4.1 Example 1: Cliff Walking

The cliff walking example is a single-agent, grid-world environment. Figure 2 presents the algorithm required for creating the example. The environment itself is created in line 2 and requires no additional parameters. An $\epsilon$-greedy exploration strategy is defined in line 5, with $\epsilon$ starting with value 1.0 and being multiplied by a decay rate of 0.99 after each episode. A Q-learning agent is defined in lines 8 and 9. The agent is defined with a learning rate $\alpha = 0.3$, a discount factor $\gamma = 0.9$, and the exploration strategy defined in line 5. The agent’s starting and ending states are also specified in the parameters to better control the end of the simulation. Finally, after all elements have been created, the simulation can be started. In lines 12 and 13, the simulation is run for 1,000 episodes. The basic output of this simulation is the reward of the agent along episodes, though other metrics can also be generated. Therefore, as seen in Figure 2, setting up a cliff walking example is simple and requires little effort.

#### 4.2 Example 2: Traffic Scenario

As for the traffic scenario, we present an example using the SUMO environment. This is a multiagent scenario, in which each agent must find a route that minimises the travel cost between its origin and destination. Observe that the agents do not know their routes a priori here. Hence, the agents may face cycles in their routes, especially in the beginning of the simulation. However, cycles tend to disappear as agents become experienced.

The algorithm required for creating and running a SUMO experiment is shown in Figure 3. The SUMO environment is created in line 2 of the algorithm, where `network.sumocfg` represents a SUMO file (which defines the road network and other aspects of the simulation), 8813 is the port through which PyRL must communicate with SUMO simulator, and `True` means that a graphical interface (of the simulation) should be displayed. For each vehicle within the scenario (line 6), a WPL agent is defined and stored in a list (lines 8 and 9). The vehicles’ properties (e.g., origin, destination) are obtained from the environment (as in line 7). No exploration strategy was specified for the WPL because it uses an internal probability vector for choosing actions. Finally, once all elements of the scenario have been created, the simulation can be started. In lines 12–13, the simulation is run for 100 episodes. The parameter used in the `run_episode` method specifies that the simulation must be halted after 50,000 steps (this is especially useful in the first episodes, where the agents have little experience and may get stuck in cycles). The basic output of this simulation is the average travel time of the agents along episodes, though other met-

---

**Figure 2**: Example of a Q-learning agent within the classic cliff walking scenario.

```python
# create a cliff walking environment
env = CliffWalking()
# define an exploration strategy
exp = EpsilonGreedy(1, 0.99)
# create a Q-learner
lrn = QLearner('AI', env, env.get_starting_state(),
               env.get_goal_state(), 0.3, 0.9, exp)
learners = []
for vID in env.get_vehicles_ID_list():
    # run 1,000 episodes
    for i in range(1000):
        env.run_episode()
env = CliffWalking()
env.run_episode(50000)
learners.append(WPL(vID, env, vDIC,
                   'network.sumocfg', 8813, True)
# create a WPL learner for each vehicle
learners = []
for vID in env.get_vehicles_ID_list():
    vDIC = env.get_vehicle_dict(vID)
    learners.append(WPL(vID, env, vDIC['origin'],
                        vDIC['destination']))
# run 100 episodes
for i in range(100):
    env.run_episode(50000)
```

**Figure 3**: Example of WPL agents that must find their routes within a SUMO environment.
rics can also be generated. As seen, PyRL facilitates the validation of RL algorithms in traffic settings, making it simple and straightforward. We remark that, even though a SUMO environment is much more complex than a CliffWalking one, the programming efforts required for setting up both environments is almost the same.

5. EXTENDING PYRL

We recall that PyRL is still under development. As such, only a few RL algorithms and validation scenarios are available. However, new modules are being developed for PyRL. Furthermore, PyRL can be easily extended with additional Environment, Learner and ExplorationStrategy modules. In fact, the ease of extending PyRL is one of its main advantages over PyBrain. Although PyBrain has more algorithms and validation scenarios than PyRL, extending it is more difficult because its code is excessively modularized. In this section, we present two simple examples on how to build a Learner and an Environment from scratch.

Our example consists in a stateless SimpleEnv environment with two actions actA and actB. The reward on actA is always 1.0 and on actB is $1/x$, with $x$ representing the number of time actB was taken. Transitions are deterministic. We also create a learner SimpleAg which chooses action uniformly at random and does not process the received reward. In other words, the agent is not actually learning.

Figure 4 presents the algorithm for creating the SimpleAg learner. We omit imports and unused abstract methods to enhance presentation. As seen, once created, the learner proceeds with the default Learner initialisation (line 3). For the ease of exposition, only the act_last and feedback_last are implemented. The act_last procedure simply checks the available actions (line 6), then chooses one uniformly at random (line 7) and, finally, returns the chosen action together with its current state (line 8). The feedback_last method, on the other hand, simply prints (line 11) the reward received for taking the action previously chosen. Observe that the learner is not actually learning, given the received reward is not being used to improve its knowledge.

The algorithm of the SimpleEnv environment is presented in Figure 5. Again, we omit imports and unused abstract methods to enhance presentation. Initially, the environment performs the default Environment initialisation (line 3) and creates an representation of the actions (line 4). Recall that actB’s reward is a function of the number of times it was taken. Thus, the counter on line 5 keeps track of this number. At each step, the environment gets the only learner from its internal list of learners (line 15). The environment then gets and prints the agent’s action (lines 16 and 17), computes the corresponding reward (lines 18–21), and provides it to the learner (line 22).

After the SimpleEnv and SimpleAg classes are defined, it remains to define the experiment and run the simulation. This is the simplest part, and is performed as in Figure 6.

6. CONCLUDING REMARKS

Validating multiagent reinforcement learning (MARL) approaches in real world problems is a challenging task. We presented PyRL – a Python Reinforcement Learning Library, which facilitates the development and validation of (MA)RL techniques. PyRL comprises different RL algorithms and validation scenarios (environments), and enables the setup of experiments with little effort. The main advantage of PyRL as compared to other libraries refers to its link with the SUMO traffic simulator, thus delivering traffic-specific environments. We presented PyRL’s architecture and discussed examples on how to use and extend it. Based on the examples, we demonstrated that setting up a complex traffic experiment in PyRL is as simple as setting up a classical cliff walking one.

PyRL is still under development. As such, our next step concerns extending PyRL with new modules. In the context of environments, we shall include a SUMO environment for managing traffic lights. As for the learners, we consider including other relevant RL algorithms/techniques.
REFERENCES


