Q-Learning Applied to OpenAI Gym

Fernando Freitas Alves
Center for Engineering, Modeling and Applied Social Sciences
Federal University of ABC
Santo André, Brazil
fernando.freitas@aluno.ufabc.edu.br

***Abstract*—Reinforcement Learning algorithms are becoming more common in research fields of science. Due to the versatility, these algorithms can be used to solve many problems that do have a closed solution and that requires an interaction between an agent and an environment. This paper shows how Q-learning is an algorithm capable of dealing successfully with this type of problems. The implementation is relatively simple and the results demonstrate accuracy in finding an optimal solution. Depending on the environment, the algorithm also showed a fast learning pace. This technique can be an entry-level for new developers of Machine Learning as well can be applicable in many problems nowadays, which includes system auto-configured and agents that learn how to play video-games.**

***Index Terms*—artificial intelligence, machine learning, reinforcement learning, Bellman's equation, Markov processes**

I. Introduction

Reinforcement Learning (RL) is becoming a popular subfield of Machine Learning (ML). Most ML techniques up to now are data-based rather than on interaction rules-based. RL is no different. More specifically, Q-learning is an RL algorithm that gathers environment data and acts against it.

The idea of RL rely on problems of agent-environment interactions (Fig. 1). An environment is the set of states and laws that dictate how actions change the current state. The agent, which most of the time is the object of study, performs that action on the environment based on its state. In turn, the environment replies to the agent with a reward and changes the state. The agent can also utilize this reward to train itself to achieve a predefined goal, which can be anything from keeping a loop of states or reaching a specific end state. The precedent events combined in order form what is called the agent-environment loop.

The RL loop described turns this field into a tool to solve problems without analyzing a given environment. Because the agent does not require prior knowledge of the environment rather than the state (or part of) it can observe, RL algorithms are data-driven. All the agent needs to care about is to process the observations and rewards from its actions. A sequence of pairs state-action is defined as a trajectory along with the algorithm steps. The trajectory is then a sub-product of a policy the agent considers based on the data it gathers. Hence, there is no need of understanding the physics and laws of the environment for an RL algorithm to solve problems.

This paper study a specific type of RL algorithm called Q-learning. Although invented more than 30 years ago by Watkins [1], Bu as shown a recent real-life application of this algorithm for an online web system auto-configuration [2]. Other works like [3] show that Q-learning is the basis of more complex techniques with better performance, like Deep$Q$-Learning where a deep neural network is used to model the Q-learning objective function. The following sections present how Q-learning works and how to build this algorithm to solve OpenAI Gym [4] environment goals.

II. Method

Q-Learning is a learning-process algorithm. The model is based on an agent, $\mathcal{A}$, loop interactions with an environment, $\mathcal{E}$. In this paper, it is expected that the environment is already implemented. Thus, the goal is to build the agent to reach a specific goal defined by a problem.

The environment contains all possible states $\mathbb{S}$ as well as tracks its current state $s \in \mathbb{S}$ within interactions. This entity is also responsible for reacting against an action $a \in \mathbb{A}$ by changing its state and providing a reward value, r, that reflects a heuristic to the given problem. The reward, in turn, is a function of the given state, the action chosen, and the next state, s', as in

$$r = R\left(s, a, s'\right). \tag{1}$$

The set of rewards as well as the state-actions rules are defined by the environment.

The agent has only access to the observable states, $o \in \mathbb{O}$ for a given state $s \in \mathbb{S} \supset \mathbb{O}$, and the rewards provided by the environment. Although the observability of the states depends on the environment and the agent, this paper will treat the observable states and the states as interchangeable, denoting both with s. So, whenever it is the case that $\mathbb{O} = \mathbb{S}$ or not, it is implied that the agent will always have access to the observable states and not the states themselves.

The agent entity reacts against the current state with an action a. The proper action is chosen from a prioritization

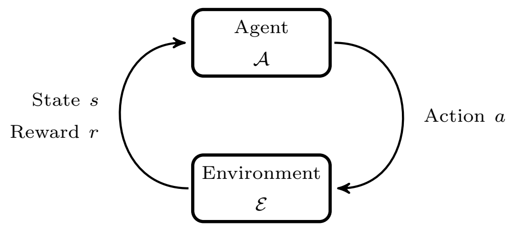


Fig. 1: Agent-environment interaction loop.

from all possible actions $\mathbb{A}$. In Q-learning, the priorities are defined as Q-values. The action for a specific state is given by a rule called policy π. The best action $a^* \equiv a^*(s)$ is then given by the optimal policy π^*. Hence, in this case, the optimal policy is the maximum value of a function $Q(s,a)$ quantified for all $a \in \mathbb{A}$ in the current state s, i.e.,

$$a^*(s) = \arg\max_{a \in \mathbb{A}} Q(s,a). \tag{2}$$

With this model defined, the goal that Q-learning algorithms tackle is find this Q function.

A. Continuous problems and the Q-table

Many problems faced by RL algorithms involve dealing with discrete variables. Say, for instance, dealing with video-games as studied by DeepMind [5], where the agent, which is usually the player, has a set of discrete actions available instead of a continuous range. The agent may have to move up, down, left, or right, for example.

Not only actions but states may also come as a group of possibilities a scenario can provide, like a given (x,y) position in a grid or a table of chess. Even if the states are continuous, the observations may be limited to be discrete. For those cases, it is common to write the function $Q(s,a)$ as a table, named Q-table.

A table of Q values is fast to update and simple to store. One can start by defining the states group as $\mathbb{S} \in \mathbb{R}^n$ for n possible states, and similarly the action group $\mathbb{A} \in \mathbb{R}^m$ for m possible actions. From there, the Q-table can be written as the $n \times m$ matrix $Q_t \equiv Q_t(s,a) \in \mathbb{R}^{n \times m}$. The goal then is to find the Q-values of this table that satisfies (2).

At last, to improve the practicality of the Q-table, even if the problem presents continuous variables, those can always be discretized. Therefore, Q-tables apply to any kind of agent-environment interaction problem like illustrated in Fig. 1.

B. The RL central problem

The Q-learning is an application of the concept of RL. However, so far the reward r was not used in its definitions. Up to this point, it was defined that the Q function prioritizes actions over states. It was also defined that the rewards are heuristic given by the environment to point how good an action is in the short term. Therefore, there is this relationship between the Q-values and the rewards. In fact, in a broader view, the function $Q^\pi(s,a)$ can be defined as an estimator of rewards under the policy π, also know as On-Policy Action-Value Function. More specifically, an estimator of the expected return, $J(\pi)$, of the sum of rewards over a trajectory τ of state-action pairs (s,a), as in

$$J(\pi) = \mathop{\mathrm{E}}_{\tau \sim \pi}[R(\tau)]. \tag{3}$$

However the policy π is optimal or not, the trajectory is the result of its rules given by the action-value function,

$$Q^\pi(s,a) = J(\pi), \tag{4}$$

which means the estimator is just an estimator. Hence, t is not known what is the optimal trajectory. However, that is exactly the central optimization problem of RL algorithms like Q-learning aim to solve, given by

$$\pi^* = \arg\max_{\pi} J(\pi). \tag{5}$$

C. The Bellman's equation and the Q-agent

All this notation is derived from Bellman's work in Dynamic Programming [6]. In his publication, he defined a solution for the optimization problem (5) that breaks it into simpler subproblems. That solution involves estimating the value of the current point in the trajectory added to the value of the agent's next move discounted by some amount γ. In other words,

$$Q^\pi(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[R(s,a,s') + \gamma \mathop{\mathrm{E}}_{a' \sim \pi}[Q^\pi(s',a')]\right], \tag{6}$$

which is know as the "Bellman's equation". Thus, when the policy is optimal, the same equation is written as

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]. \tag{7}$$

The so-called Q-agent algorithm then approximates the optimal action-value function $Q^*(s,a)$ by direct small updates. These updates act on iteration of the Q-values over a function $Q_t(s,a)$ at time t:

$$q_t = R(s,a,s') + \gamma \max_{a'} Q_t(s',a'). \tag{8}$$

This Q-learning algorithm, proposed by Watkins [1], keeps an estimate $Q_t(s,a)$ of $Q(s,a)$ for each state-action pair. Furthermore, according to Szepesvari, when Q_t is close to Q^*, the policy that greedy choose the action with higher Q-value will be close to optimal [7].

D. Algorithms

Finally, this paper demonstrates the development of algorithms to implement the Q-agent. Although some function names are derived from OpenAI Gym [4], the syntax is generic and does not require that library to function.

To start with, it is necessary to define a general procedure of the agent-environment interaction loop for RL problems (Fig. 1), as in Algorithm 1. Each iteration utilized three functions from the environment $\mathcal{E}$ given by the library:

- RESET($\mathcal{E}$): a reset function called once each episode. It provides the initial state $s_0 \in \mathbb{S}$.
- DONE($\mathcal{E}$): a done function called at every iteration. It indicates if the simulation is over.
- STEP($\mathcal{E}, a$): a step function called at every iteration. It provides a set of state-reward (s-r) for a given action a.

The other two functions are defined by the agent, which implementation is the goal of this paper: ACT($\mathcal{A}, s$) and TRAIN($\mathcal{A}, s, a, s', r$). The current algorithms for both functions are based on the discrete formulation of the $Q_t(s,a)$, described as Q-table in section II-A.

The act function is responsible for returning the best action for a given state by using the current Q-table, as in Algorithm 2. To remove bias from previous Q-values that may lead the solution to a locally optimal policy, also known as

Algorithm 1 RL general algorithm

```
function ReinforcementLearning($\mathcal{E}, \mathcal{A}$)
Input: Environment $\mathcal{E}$, agent $\mathcal{A}$
Output: Trained agent $\mathcal{A}$
 1: for all $episodes$ do
 2:     $s_0 \leftarrow$ Reset($\mathcal{E}$)
 3:     $s \leftarrow s_0$
 4:     $r \leftarrow 0$
 5:     while not Done($\mathcal{E}$) do
 6:         $a \leftarrow$ Act($\mathcal{A}, s$)
 7:         $s', r \leftarrow$ Step($\mathcal{E}, a$)
 8:         $\mathcal{A} \leftarrow$ Train($\mathcal{A}, s, a, s', r$)
 9:         $s \leftarrow s'$
10:     end while
11: end for
12: return $\mathcal{A}$
```

Algorithm 2 Q-Agent act algorithm

```
function Act($\mathcal{A}, s$)
Input: Agent $\mathcal{A}$, state $s$
Parameters: Agent $\mathcal{A} \rightarrow$ table of state-action pair values $Q_t$,
    exploration rate $\epsilon$
Output: Action $a$
 1: $\widetilde{Q} \leftarrow$ sample $m$ random numbers
 2: $\widetilde{Q}_t(s,a) \leftarrow Q_t(s,a) + \epsilon\widetilde{Q}$
 3: $a \leftarrow \arg\max_a \widetilde{Q}_t(s,a)$
 4: return $a$
```

exploitation, it is common to allow agents to explore other options rather than the provided from current policy. The new options can be any random decision, which is simulated by a random $\widetilde{Q}$ value added to the current Q-table. That balance between exploration and exploitation is controlled by a number ϵ between 0 and 1 called exploration rate. The return value is defined by (2). If the optimization is finished, the $\arg\max_a Q_t(s,a)$ represents the optimal policy π^* from (5) when no exploration is done.

The train function, on the other hand, is the implementation of the Q-learning optimization process defined by (8). The idea is basically to update the current iteration of the Q-table towards the optimal solution, as in Algorithm 3. First, the current small step q_t of the Bellman's equation (6) is calculated. That value is the objective function of the optimization problem. Then, the Q-table is updated against the loss, which is the difference between the objective function and the current Q-value for a given state-action pair. The discount factor is not necessarily constant. In this paper, it has an update rule equal to a decaying constant real number between 0 and 1 that multiplies it at every call to the train function. That choice of the update was made to faster convergences, even though it can bias the results. Given enough time, if the discount decaying rate is sufficiently small, the table will become a representation of the optimal Bellman's solution (7).

The implementation of the algorithms in Python program-

Algorithm 3 Tabular Q-learning training algorithm

```
function Train($\mathcal{A}, s, a, s', r$)
Input: Current state $s$, action chosen $a$, next state $s'$, reward $r$
Parameters: Agent $\mathcal{A} \rightarrow$ table of state-action pair values $Q_t$,
    discount factor $\gamma$, learning rate $\alpha$
Output: Agent $\mathcal{A}$ with updated table of state-action pair
    values $Q_{t+1}$
 1: $q_t \leftarrow r + \gamma \max_{a'} Q_t(s', a')$
 2: $e \leftarrow q_t - Q_t(s,a)$
 3: $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha e$
 4: return $\mathcal{A}$
```

TABLE I: Environments variables

Name	**Observation space**	**Action space**	**Reward space**
FrozenLake-v0	Discrete	Discrete	$\{0,1\}$
FrozenLakeNoSlip-v0	Discrete	Discrete	$\{0,1\}$
MountainCar-v0	Continuous	Discrete	$(-\infty,\infty)$
MountainCarContinuous-v0	Continuous	Continuous	$(-\infty,\infty)$

ming language is available at the repository https://github.com/fernando-freitas-alves/reinforcement-learning.

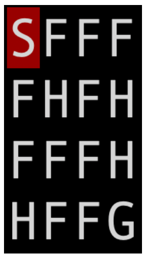
III. Results

The algorithms were tested against 4 environments from OpenAI Gym library [4], were one was modified:

- **FrozenLake-v0**: "The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile." [4]
- **FrozenLakeNoSlip-v0**: A external modified version of the FrozenLake-v0 by Shawn where the randomness of the simulated slippery was removed. Hence, this is a deterministic environment. [8]
- **MountainCar-v0**: Originated from Moore's Ph.D. thesis [9], this environment is a simulated car in a one-dimensional track, positioned between two hills. The goal is to make the agent drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Hence, the car is forced to drive back and forth to build up momentum.
- **MountainCarContinuous-v0**: A similar problem to the MountainCar-v0 where the only difference is a continuous action space. In other words, the left and right movements are real numbers simulating a gas pedal. Besides, the reward is greater if the agent spends less energy to reach the goal.

Table I summarizes the major variables for each environment, while Fig. II and III illustrate their states.

The results are shown as 3 curves by episodes: cumulative goals reached, rewards, and exploration rate; for each environment simulation. The cumulative goals reached shows the total



S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

TABLE II: Initial state of the environments FrozenLake-v0 and FrozenLakeNoSlip-v0. The letters S, F, H, and G represents the start position, frozen water, holes, and the goal. The red background selected letter represents the current state. The agent cannot move outside the grid.

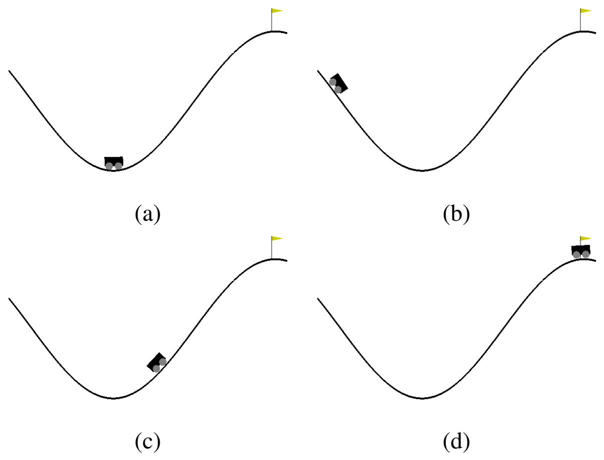


TABLE III: Some of the possible states of the environments MountainCar-v0 and MountainCarContinuous-v0. The objective is to the car to reach the right end of the screen. If the agent move outside the left end of the screen, the simulation is terminated as failed. The starting pointing is in a random position near the state Fig. IIIa.

of how many times the goal of the environment was achieved by the agent within the limit steps preconfigured by the library. Similarly, the reward shows the return of the environment, while the exploitation rate shows the evolution of that variable over episodes.

Fig. 2 shows the results for FrozenLakeNoSlip-v0. Since this is a discrete deterministic environment, the states are updates in a Markov chain. The curves demonstrate constant cumulative goals reached slope and the fast saturation of the rewards. This presents the Q-learning algorithm was not only able to succeed in most of the episodes but also had fast training to the optimal solution.

On the other hand, the results for FrozenLake-v0 in Fig. 3 shows a slower training and lower efficacy. The constant cumulative goals slope was lower and the rewards kept oscillating around a plateau after some training. This demonstrates that a stochastic simulation does not have as fast and as an optimal solution as the one finds for the same environment but deterministic. Adding a random factor to the next states after an action increased the difficulty level to the algorithm to find a correlation between his actions and high rewards. That randomness changed the optimal policy in a way that it becomes unpredictable to find the best next action. Although this problem could be found by any RL algorithm, Q-learning showed it is capable of finding a consistent solution that solves the problem 40% of the time with the given stochasticity.

Changing to a continuous state environment, the results for MountainCar-v0 are shown in Fig. 4. The discretization of the position and velocity states took 20 numbers for each. The curves were similar to a mix between the ones from the FrozenLake. On one hand, the return found a plateau and oscillated randomly around it after some episodes. This illustrates stability in the learning process, where the Q-table is not so largely updated as in previous episodes. Intriguingly, the rewards had a sudden decay followed by a fast recuperation after a few episodes. That is mainly due to unvisited states and exploration, which can be lower but never null. On the other hand, even though the table stabilized but kept a random variation, the cumulative goals reached had a high slope. Indeed, most of the episodes ended successfully, demonstrating the solution found is close to optimal within the given episodes to train.

Finally, the results for the similar but totally continuous MountainCarContinuous-v0 problem is shown in Fig. 5 and 6. The first shows the result when the states have 20 discrete values and the actions have only 3. This is an equivalent problem from MountainCar-v0, since all actions were mapped to left, right, and nothing. Despite their similarities, they are still different in physics simulation, since now the car has a rule that considers the gas pedal. Because of that, the decaying constant of the exploration rate had to be set to a smaller value. Nonetheless, the results showed also a plateau for return but with smaller variations, while the stabilized found policy was optimal to the point of almost all episodes converging, save for a few still in training. When comparing with Fig. 4, both cumulative goals reached slopes are equivalent, showing the solution found was similar.

The second figure shows the results when the states and the actions have the same 20 discrete values. The convergency took considerably more episodes to happen, but the slope of the cumulative goals reached was also similar to Fig. 4. One can consider that, due to a higher decaying constant of the exploration rate, the training requires more steps to converge, since more actions will be chosen randomly. In any case, the return and accuracy were similar to a lower-discretized problem, showing that raising the numbers of possible states

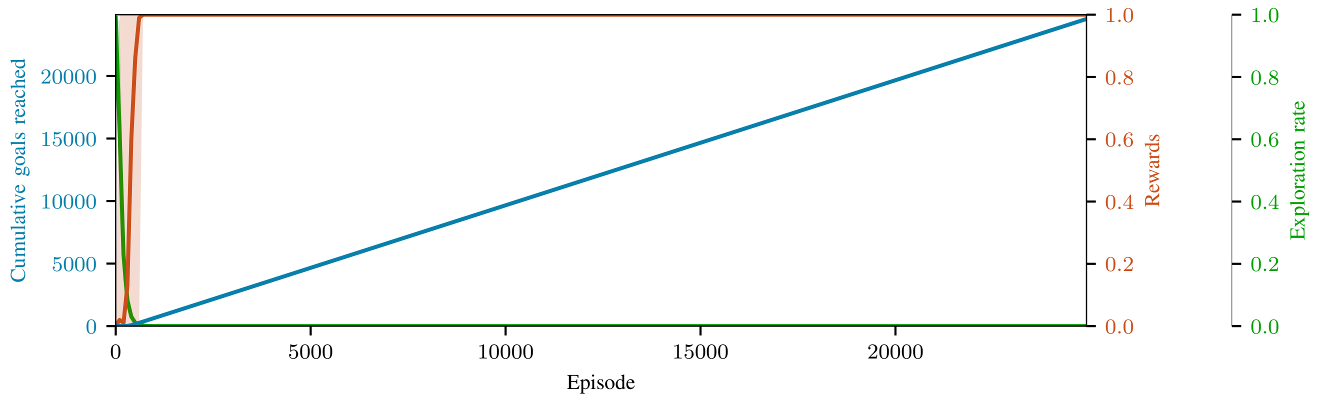


Fig. 2: Results from FrozenLakeNoSlip-v0 environment.

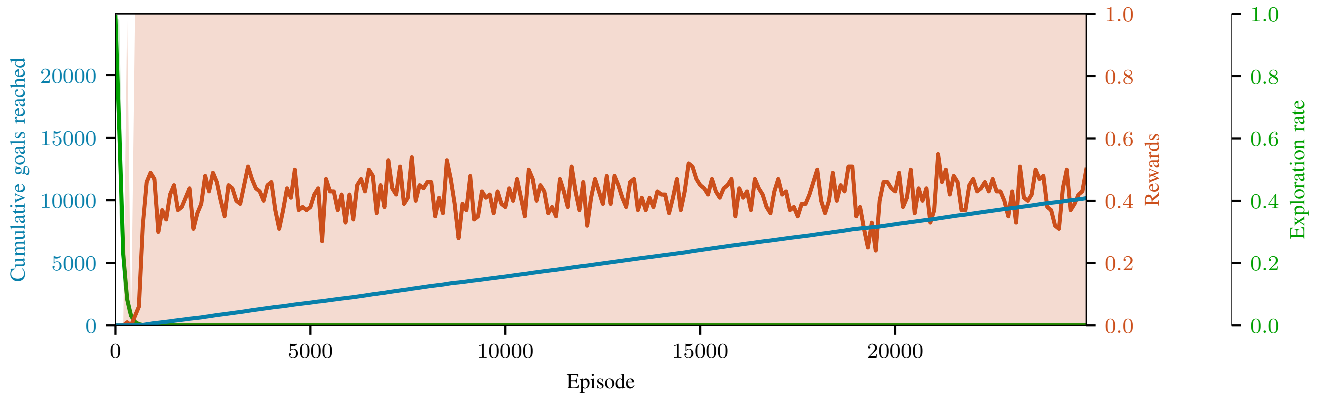


Fig. 3: Results from FrozenLake-v0 environment.

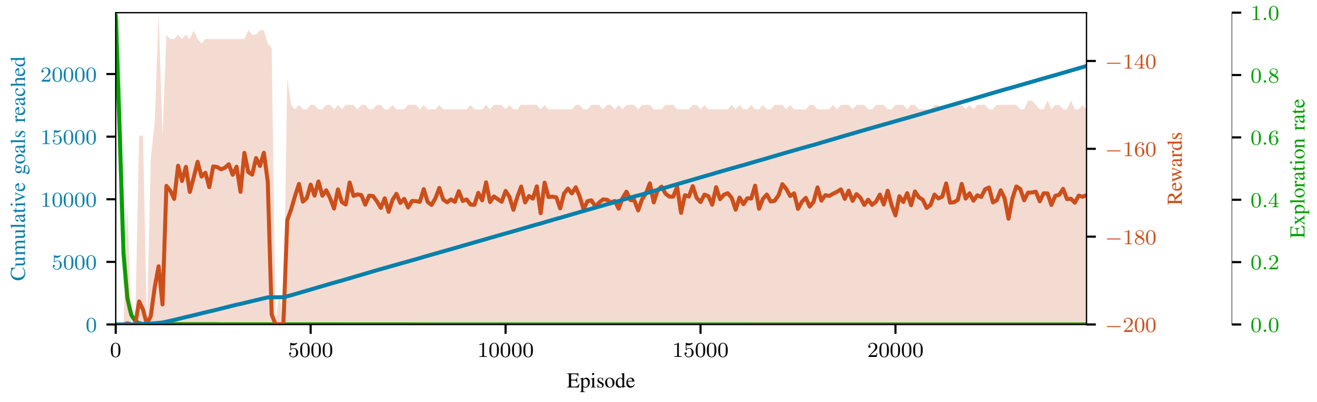


Fig. 4: Results from MountainCar-v0 environment.

among a continuous environment does not necessarily make the algorithm more efficient. The higher the discretization, the more Q-values the table will have to keep and, consequently, the more system memory the agent will need.

IV. Conclusion

Q-learning exhibited successfully results in the RL optimization problem. Moreover, RL showed to be a feasible and robust technique to solve agent-environment problems. The implementation of the algorithm is relatively simple and results in fast convergencies in some cases. However, this method requires some fine-tuning of the parameters that may take some time to find a quick training for the given environment.

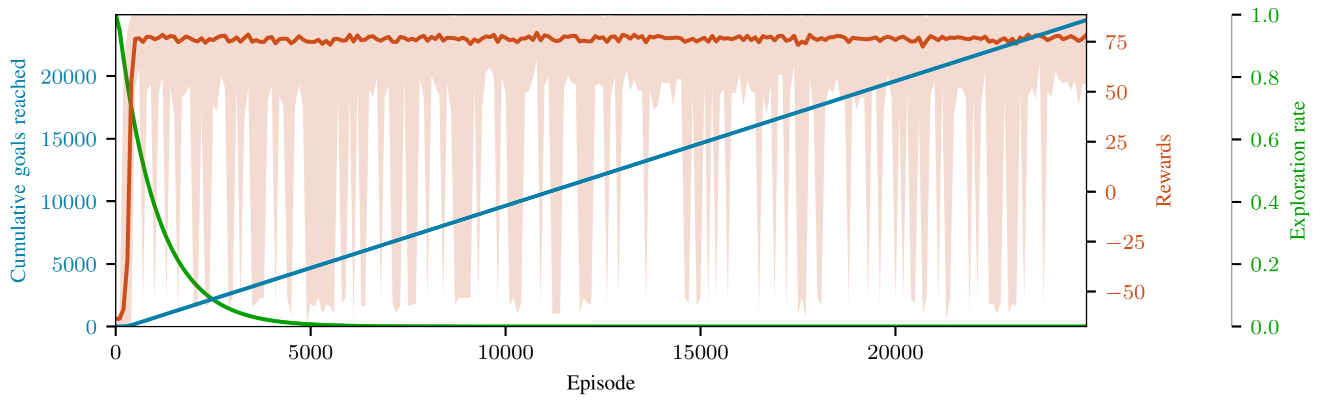


Fig. 5: Results from MountainCarContinuous-v0 with 3 discretized action steps environment.

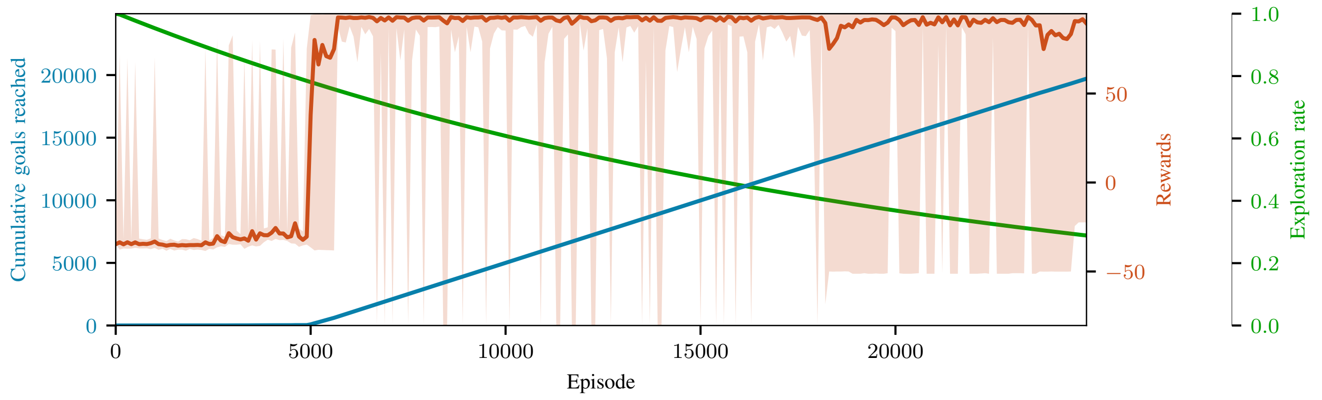


Fig. 6: Results from MountainCarContinuous-v0 with 20 discretized action steps environment.

Due to that limitation, only 4 environments were tested in this paper. The results also showed that the developer of the agent model must choose carefully the discretization level of the problem, in case it has continuous variables. In summary, the Q-learning algorithm is a good tool for practicing and can serve as an entry-level for RL. However, more practical tools, like Deep Q-Learning, can be developed to provide the same results more efficiently.

References

[1] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Oxford, 1989.

[2] X. Bu, J. Rao, and C. Z. Xu, "A reinforcement learning approach to online web systems auto-configuration," in *Proceedings - International Conference on Distributed Computing Systems*, 2009.

[3] G. Zheng, F. Zhang, Z. Zheng, Y. Xiang, N. J. Yuan, X. Xie, and Z. Li, "DRN: A Deep Reinforcement Learning Framework for News Recommendation," *[WWW2018]Proceedings of the 2018 World Wide Web Conference*, 2018.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016, cite arxiv:1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[6] R. Bellman, *Dynamic Programming*. Dover Publications, 1957.

[7] C. Szepesvári, "Algorithms for reinforcement learning," 2010.

[8] S. M., "Code from 'Intro to OpenAI Gym' tutorial video," https://github.com/the-computer-scientist/OpenAIGym/blob/7be80647e6e\090c76f28ea03b7d1ba891db75f2f/QLearningIntro.ipynb, Oct. 2018.

[9] A. W. Moore, "Efficient memory-based learning for robot control," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-209, Nov. 1990. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf