Considerations on Implementing Q-Learning Algorithm for Robot Path Planning

Andrei DUTCEAC

***Abstract*—Artificial Intelligence has made remarkable advancements, becoming increasingly potent and expanding exponentially across multiple domains such as healthcare, robotics, finance, and autonomous vehicles. This paper focuses on introducing a fundamental algorithm in reinforcement learning, specifically tailored for path planning—an area of great interest for autonomous robots and self-driving cars. The algorithm incorporates various models and demonstrates versatile applications. To enhance its effectiveness, we have introduced additional functions to mitigate random actions taken by the agent, thereby reducing the occurrence of suboptimal paths and minimizing the time required to find an optimal solution.**

***Index Terms*—action, reward, state, *Q* table, policy.**

I. Introduction

Reinforcement learning is one of the most active areas of research in artificial intelligence, which has found practical applications in diverse fields, including robotics, game playing, recommendation systems, finance, and healthcare. Its ability to learn optimal behavior in dynamic and uncertain environments holds great promise for building intelligent systems that can adapt and make autonomous decisions.

The goal of the agent is to learn an optimal policy—a strategy that maximizes its long-term reward. Unlike supervised learning, where the agent is provided with labeled examples, or unsupervised learning, where it discovers patterns in unlabeled data, reinforcement learning relies on trial and error.

Reinforcement learning algorithms employ various techniques to tackle these challenges [1]. They often use a value function or a policy to guide decision-making. The value function estimates the expected future rewards, helping the agent evaluate the potential of different actions. The policy, on the other hand, determines the mapping from states to actions, indicating the best course of action in a given situation.

Robot path planning is a fundamental problem in robotics that involves determining a collision-free path for a robot to move from its initial position to a desired goal position. It is an essential task in various robotic applications, including autonomous navigation, industrial automation, and mobile robotics. The goal of path planning is to find an optimal or near-optimal path that satisfies certain criteria, such as minimizing distance traveled, avoiding obstacles, considering robot dynamics, and accounting for any specific constraints or requirements of the robot and its environment. There are many path-planning algorithms being studied and continuously developed, starting with the Dijkstra algorithm [2], artificial neural networks to genetic algorithms.

II. Q-Learning Algorithm

Q-Learning was proposed by Watkins in 1992 [3] and is based on the interaction between an agent in some defined environment. The goal for the agent is to learn how to make intelligent decisions when interacting with the environment by performing an action and moving from one state to another. States are the positions or moments that the agent can be in and, based on the action he makes, the agent receives a reward which is a numerical value, usually a positive number for a good action and a negative number for a bad action. For example, in the Snake game, the agent who represents the snake is performing actions (up, down, left, right) in order to catch food and receive rewards (points). A good action is catching the food and a bad action is reaching his tail which is rewarded with the end of the game. So the main goal for the agent is to learn how to catch as much food as he can and trying not to reach his tail.

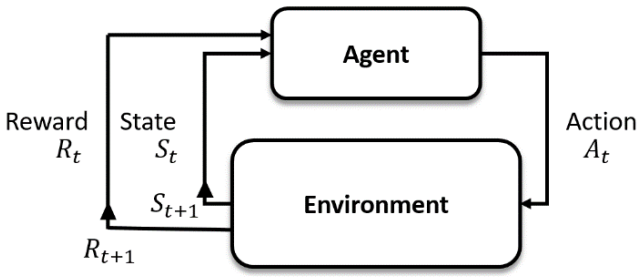

Figure 1. Action-reward feedback loop

Another important element is the *Q* table. In this table, the rows represent all the states in which the agent can perform actions and the columns contain the values for each action the agent performs for the corresponding state. Initially, all the values from the table are 0 or randomly chosen, and with each reward that the agent receives the table is updated with positive or negative values. The purpose of this table is to extract an optimal path for the agent or an optimal policy which is another important concept in reinforcement learning.

A policy defines the agent's behavior in an environment and tells the agent what action he should perform in each state, in correspondence with our table the policy will be represented by the actions which have the maximum values for each individual state. These values will be obtained using the Bellman equation. Named after Richard Bellman, the equation is a fundamental concept in dynamic programming

A. DUTCEAC is with the Military Technical Academy "Ferdinand I", Bucharest, Romania (e-mail: andrei.dutceac@mta.ro)

Digital Object Identifier 10.32754/JMT.2023.1.07

and reinforcement learning. It describes the relationship between the value of a state and the values of its neighboring states. In the context of Markov Decision Processes (MDPs), the Bellman equation defines the value function, which represents the expected cumulative rewards an agent can obtain from a particular state under a given policy. The Bellman equation (1) for the *Q* value state-action pair can be written as follows:

$$Q(s,a) = R(s,a,s') + \gamma * Q(s',a') \quad (1)$$

In the above equation, $Q(s,a)$ implies the value of the current state when performing action a, $Q(s',a')$ is the value of the next state-action pair, R implies the immediate reward gain while performing an action to move from state s to s', and γ represents the discount factor. This factor helps us to prevent the return from reaching infinity by adding unlimited rewards. By setting a specific value we decide how much importance we give to the immediate rewards or to the future rewards. This value ranges from 0 to 1 so when we set a small value close to 0, it implies that we give importance to immediate rewards and when we set it close to 1 we give more importance to future rewards. The manner in which we select the action is another crucial aspect to consider when utilizing the equation and for this, we use the epsilon-greedy algorithm. Epsilon-greedy is a commonly used strategy for selecting actions in *Q*-learning, which is a popular algorithm for reinforcement learning. Epsilon-greedy balances exploration and exploitation by randomly selecting a random action with probability epsilon (ε) and selecting the action with the highest *Q*-value with probability 1 – ε.

Greedy-epsilon strategy

if random(0, 1) < ε
    perform random action a
else
    choose action a with maximum *Q* value for $Q(s,a)$
end if

With probability ε, we choose an action (a), uniformly at random from the set of otherwise possible actions (with probability 1 – ε), select the action (a) with the highest *Q*-value for the current state $Q(s,a)$. The epsilon-greedy strategy ensures that the agent explores the environment by randomly choosing actions, especially in the early stages of learning when the *Q*-values are uncertain. As the learning progresses and the *Q*-values become more accurate, the agent gradually shifts towards exploiting the actions with higher *Q*-values, focusing on the best-known actions. With these, we can write the updated form of the Bellman (2) equation as follows:

$$Q(s_t,a_t) = (1-\varepsilon) * Q(s_t,a_t) + \varepsilon * \left(r + \gamma * \max_a Q(s_{t+1},a) \right) \quad (2)$$

The Bellman equation shows that the value of a state is equal to the maximum expected sum of immediate reward and the discounted value of the next state, averaged over all possible next states (weighted by their transition probabilities). The agent seeks to choose actions that maximize this value function to make optimal decisions. The pseudo-code of Q-Learning algorithm is shown as follows.

Q-Learning Algorithm

```
Initialize Q table, ε, max_steps
for e=1 in episodes
    Initial state s_t
    for s=1 in max_steps
        select action a
        use (2) to obtain reward r and next state s_{t+!}
        decrease ε
    end for
end for
```

The premise and foundation of path planning lie in environmental information [4]. The environment represented through maps, grids, or other forms, serves as the essential input for the path-planning process [5]. The grid method is one of the most widely used methods in path planning research. Grid world refers to a discrete environment that is represented as a grid or a matrix of cells. Each cell in the grid can represent a state or a location in the environment. Grid worlds are often used to simulate simplified environments, such as a maze, where an agent can move between cells. Each cell may have certain properties or characteristics, such as obstacles, rewards, or costs associated with moving through them [7]. Agents typically navigate through the grid world by taking actions (e.g., moving up, down, left, or right) to transition between cells [8].

Fig. 2 shows the path planning environment for this paper. With a size of 9 × 9, the black square represents the obstacle, the robot's position represents the starting point and the flag represents the endpoint. The agent needs to find the path from the starting point to the endpoint with minimum cost and avoid obstacles.

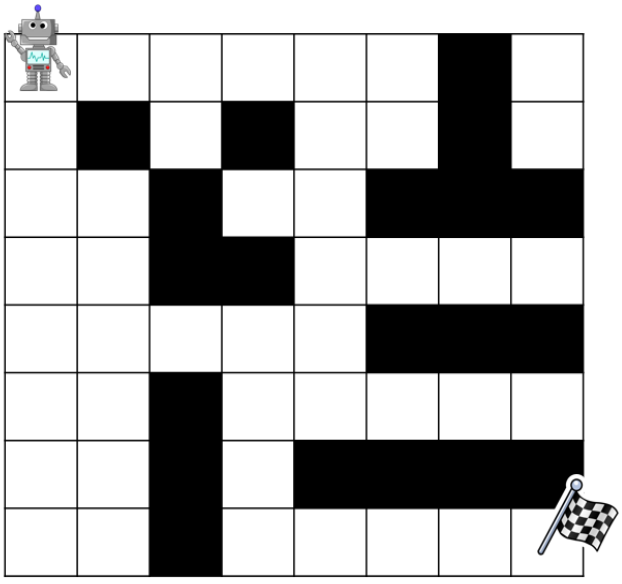

Figure 2. Grid environment

III. THE IMPROVED ALGORITHM

To reduce the frequency of random actions in the *Q*-Learning algorithm, we suggest augmenting the initial approach with a two-step method. Currently, the choice of action 'a' for transitioning between states is determined by alternating between *Q* table values and selecting a random

 Digital Object Identifier 10.32754/JMT.2023.1.07

possible action. This can lead to a scenario where, if an action results in a negative reward at a particular state, there's a chance that the agent will repeatedly choose that action when passing through that state, incurring additional costs for the overall path. Hence, our proposed solution involves incorporating a 2-step process into the existing algorithm to mitigate the excessive use of random actions.

Proposed Algorithm

Initialize action_table, action_history
Initialize Q table, ε
for e=1 in episodes
    Initial state s_t
    for s=1 in max_steps
        if random < ε
            choose random action a_t
        else
            a_t = max($Q(s_t, a_t)$)
            if $a_{t-1} = a_{t-2}$
                $a_t = a_{t-1}$
        end if
        use (2) to obtain reward r and next state s_{t+1}
        add action a_t to action_history vector
        if reward r < 0
            delete action a_t from action_table
        end if
        decrease ε
    end for
end for

To enhance the learning process, the first step is to introduce an additional table containing all possible actions for each state. When the agent performs an action resulting in a negative reward, a precautionary measure is taken to prevent the agent from repeating the same unfavorable experience in subsequent episodes. This involves removing the action associated with the negative reward from the action table. Consequently, when the agent encounters the same state again, it selects one of the remaining available actions from the table. This approach effectively helps the agent avoid negative rewards and facilitates more efficient training by reducing the number of steps required. As a result, the learning process experiences a substantial improvement in performance and effectiveness.

In the subsequent step, a vector is introduced to retain the action history from previous states. This enables the agent to observe and identify certain paths or patterns that can be followed. In continuous environments, the agent typically tends to move forward and maintain its current direction, deviating only when obstacles are detected. This concept can be applied in grid world scenarios to encourage the agent to consistently follow a particular direction. By periodically examining the action history, if the agent's last two actions indicate the same direction, the agent will opt for the same action in the subsequent state. This approach facilitates efficient movement as it allows the agent to traverse the environment even if it initially heads in an unfavorable direction, ultimately returning to the same trajectory.

IV. Simulations Results

For testing the algorithm we used Python language and PyCharm IDE to create and simulate the environment [6]. The rewards chosen for this grid world are -1 point if the agent hits the obstacles and 20 points for reaching the endpoint. In Fig. 3 and 4 we simulate the initial algorithm on 100 episodes with the maximum steps per episode being 1000.

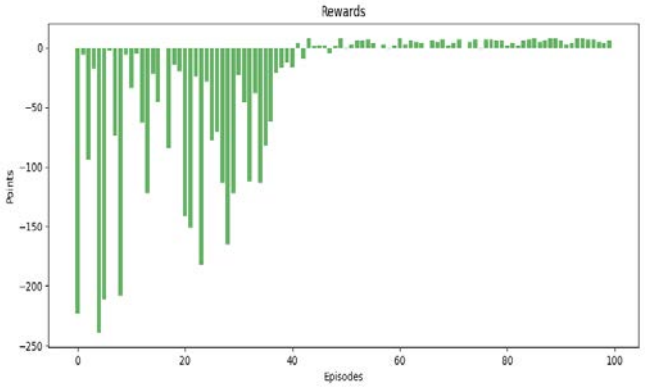

Figure 3. Q-Learning episode rewards

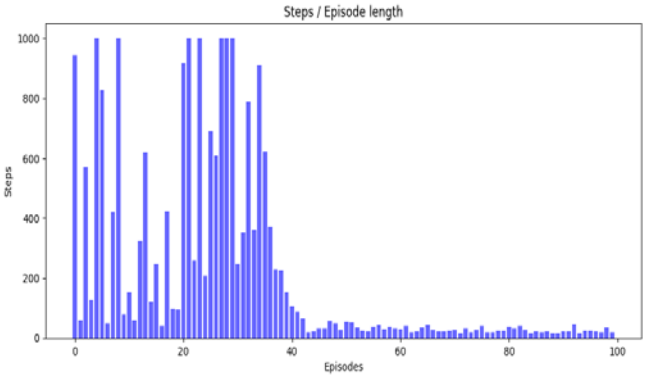

Figure 4. Q-Learning steps /episode

Fig. 3 illustrates the progression of total rewards per episode throughout the training phase. In the initial phase, due to the high value of the exploration rate (ε), the agent predominantly selects actions randomly. This is evident from the negative reward values that plummet to –200. As the training progresses, the exploration rate gradually decreases, prompting the agent to shift its focus toward selecting optimal actions rather than random ones.

Consequently, in the latter part of the graph, we see the transition of the negative rewards to positive values. This shift signifies that the agent is able to navigate to the endpoint more swiftly without encountering obstacles, resulting in a more favorable overall reward accumulation. Also, Fig. 4 illustrates the evolution of the steps per episode ranging from the maximum number of steps to the shortest path of 14 steps. There are 2 ways for the agent to reach the endpoint, he can choose to move down or to move right and find the path. In both cases he will move randomly and receive negative rewards until he learns to avoid them and progresses towards attaining positive rewards.

We can mention here that for the episodes that reach 1000 steps, the agent did not reach the endpoint. In the next figures, we applied the proposed algorithm to compare the results.

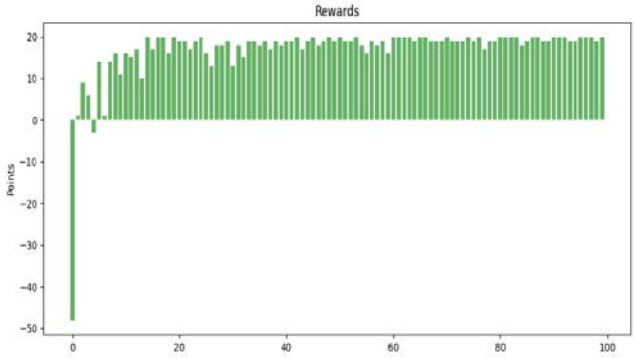

Figure 5. Improved algorithm rewards

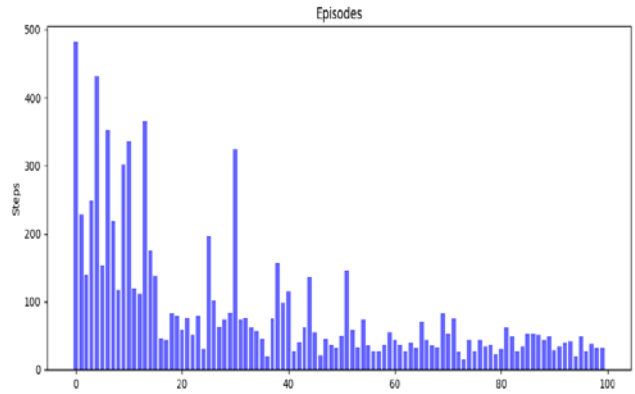

Figure 6. Improved algorithm steps/episode

Fig. 5 shows an important improvement by eliminating the actions with negative rewards so now we have only 2 negative rewards and a maximum negative reward much smaller than the previous one. Fig. 6 implies the reduced cost of episodes, this time no episode being longer than 500 steps. It's important to mention that in both situations we use exponential decay for reducing the exploration rate, shown in Fig. 7. The exploration rate determines the balance between exploration (taking random actions to discover new states) and exploitation (selecting actions based on learned knowledge) in the *Q*-learning algorithm. By reducing epsilon over the course of training, the agent becomes more focused on exploiting its learned *Q*-values and making optimal decisions.

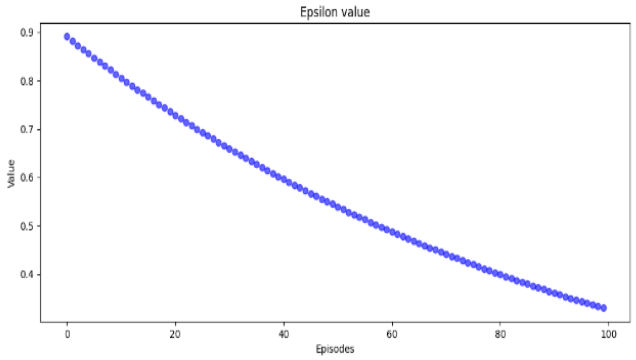

Figure 7. Epsilon-decay method

V. CONCLUSION

This paper applies the improved *Q*-Learning algorithm to a classic grid environment and shows that by reducing the number of random actions the agent will move more efficiently and receive more rewards. Implementing epsilon decay in *Q*-learning involves updating the exploration rate at the end of each episode or time step according to the chosen decay method. This ensures that the agent gradually shifts from exploration to exploitation as it learns and improves its policy in the given environment.

In the context of the grid world, path planning often involves finding the shortest path between two points or finding an optimal path that maximizes a certain objective, such as minimizing time, energy consumption, or risk.

REFERENCES

[1] C. J. Watkins and P. Dayan, "Technical note: Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, May 1992.
[2] M. Parimala, S. Jafari, M. Riaz, M. Aslam, "Applying the Dijkstra algorithm to solve a linear diophantine fuzzy environment", vol. 13, no. 9, p. 1616, Sep. 2021.
[3] S. Ravichandiran, "Deep Reinforcement Learning with Python", Ed. Packt Publishing, UK, 2022.
[4] B. Jang, M. Kim, G.Harerimana, J. Kim "Q-Learning Algorithms: A Comprehensive Classification and Applications", Sep. 2019, doi: 10.1109/ACCESS.2019.2941229
[5] C. Wang, X. Yang, H. Li, "Improved Q-Learning Applied to Dynamic Obstacle Avoidance and Path Planning," Aug. 2022, doi:10.1109/ACCES.2022.3203072
[6] G. Leekha " Learn AI with Python" , *BPB Publications*, India, 2022
[7] I. Maurovic, M. Seder, K. Lenac, and I. Petrovic, "Path planning for active SLAM based on the D∗ algorithm with negative edge weights", *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 48, no. 8, pp. 1321–1331, Aug. 2018.
[8] E. S. Low, P. Ong, and K. C. Cheah, "Solving the optimal path planning of a mobile robot using improved Q-learning", *Robot. Auto. Syst.*, vol. 115, pp. 143–161, May 2019.

 Digital Object Identifier 10.32754/JMT.2023.1.07