Reinforcement Learning: Experimental Study of Q-Learning Applied to Labyrinth Problem

Mehyar MLAWEH, Nadia BENYOUSSEF
mehyarmlaweh0@gmail.com, nadia.benyoussef@dauphine.tn
Paris Dauphine University-PSL

Abstract

This report explores the application of Q-learning, a reinforcement learning algorithm, to solve Labyrinth navigation problems using intelligent agents. Beginning with an introduction to Markov Decision Processes, it outlines the theoretical framework necessary for understanding sequential decision making in uncertain environments. The Q-learning algorithm is then introduced, along with its operation and key parameters. Subsequently, an implementation of Q-learning in Python for labyrinth-solving is presented, including details on the maze environment and algorithmic considerations. The report also discusses experiments varying exploration-exploitation strategies and labyrinth sizes, providing insights into their impact on solution quality. Results highlight the effectiveness of Q-learning in labyrinth navigation and underscore its potential in addressing complex problems through reinforcement learning techniques. **Keywords** – Q-learning — Maze Navigation — Reinforcement.

1 Introduction

In this report, we're exploring the effectiveness of Q-learning in solving maze puzzles. We'll begin by providing a brief overview of key concepts in reinforcement learning, such as Markov Decision Processes and the Q-learning algorithm itself. Using Python and libraries like OpenAI Gym and Pygame, we'll develop a program that learns to navigate through mazes. Our primary objective is to understand the impact of different Q-learning settings—such as learning rate, exploration rate, and maze size—on the speed and quality of solutions. Through systematic experimentation and parameter tuning, we aim to gain insights into the optimal configuration of Q-learning for addressing challenging real-world problems.

2 Reinforcement learning concepts

For reinforcement learning problems the learner is called the agent. The agent is the decision maker within an environment. The environment is everything the agent interacts with. The agent is selecting possible actions provided by the environment which also gives feedback to the agent. Feedback consists of rewards defined by the environment which are given upon change of state. A state is often an abstract representation of the environment [Sutton and Barto, 2015].

3 Markov Decision Processes

3.1 What is it?

A Markov Decision Process (MDP) is a mathematical framework[Wikipedia, 2024] used in artificial intelligence and decision-making. It helps to model and solve problems where an agent makes decisions in an uncertain environment. The MDP is based on the concept of a Markov chain, which is a sequence of events where the probability of each event depends only on the state of the previous event.

3.2 Components

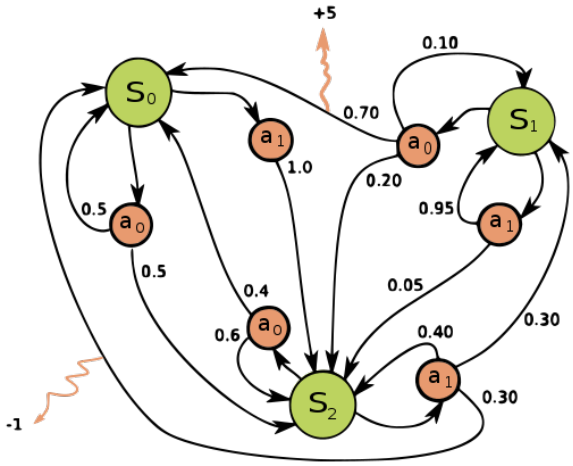

Figure 1: Example of a simple MDP

A MDP is a 4-tuple (S, A, P_a, R_a) [Bellman, 1957], where:

- S is a set of states called the state space,
- A is a set of actions called the action space (alternatively, A_s is the set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.

The state and action spaces may be finite or infinite, for example the set of real numbers. Some processes with countably infinite state and action spaces can be reduced to ones

with finite state and action spaces.

The graphical representation of the MDP model is as follows: The MDP model uses the **Markov Property**[Markov, 1954],

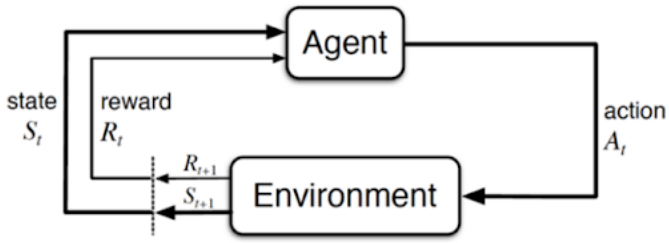

Figure 2: Graphical representation of MDP

which states that the future can be determined only from the present state that encapsulates all the necessary information from the past. The Markov Property can be evaluated by using this equation:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, S_3, \ldots, S_t] \tag{1}$$

According to this equation, the probability of the next state ($P[S_{t+1}]$) given the present state (S_t) is given by the next state's probability ($P[S_{t+1}]$) considering all the previous states ($S_1, S_2, S_3, \ldots, S_t$). This implies that MDP uses only the present/current state to evaluate the next actions without any dependencies on previous states or actions.

3.3 Policy and Value

A **policy** (π) describes the decision-making process of the agent. In the simplest case, the policy for each state refers to an action that the agent should perform in that state. This type of strategy is called deterministic policy. Each state is assigned an action, for example for state s_1: $\pi(s_1) = a_1$. A deterministic policy can be displayed in a table, where an action can be selected in different states: [Dittert, 2020]

State (s)	Action (a)
s_1	a_1
s_2	a_3
s_1	a_2
s_2	a_2

In general, a policy assigns probabilities to every action in every state, for example $\pi(s_1|a_1) = 0.3$. The policy thus represents a probability distribution for every state over all possible actions. Such a policy is called a stochastic policy. In a stochastic policy, several actions can be selected, whereby the actions each have a probability of non-zero and the sum of all actions is 1. Thus, it can be said that the behavior of an agent can be described by a policy, which assigns states to a probability distribution over actions. The policy is only dependent on the current state and not on the time or the previous states.

The **Value Function** represents the value for the agent to be in a certain state. More specifically, the state value function describes the expected return G_t from a given state. In general, a state value function is defined concerning a specific policy, since the expected return depends on the policy:

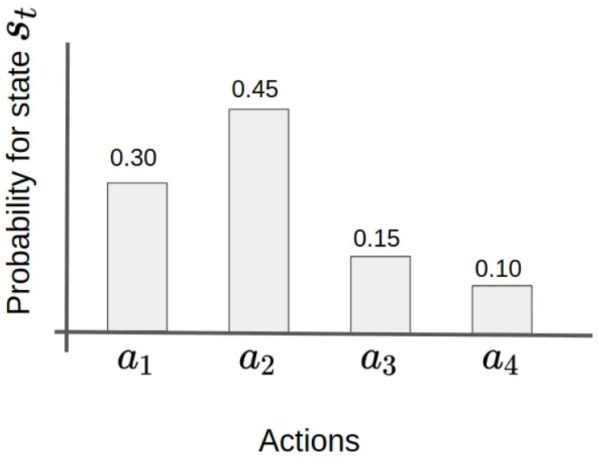

Figure 3: Probability distribution for a stochastic policy [Dittert, 2020]

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \,|\, S_t = s] \tag{2}$$

The index π indicates the dependency on the policy. Furthermore, an action-value function can be defined. The action-value of a state is the expected return if the agent chooses action a according to a policy π.

$$q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t \,|\, S_t = s, A_t = a] \tag{3}$$

Value functions are critical to Reinforcement Learning (RL). They allow an agent to query the quality of his current situation rather than waiting for the long-term result. This has a dual benefit. First, the return is not immediately available, and second, the return can be random due to the stochasticity of the policy as well as the dynamics of the environment. The value function summarizes all future possibilities by averaging the returns. Thus, the value function allows an assessment of the quality of different policies.

A fundamental property of value functions used throughout RL is that they satisfy recursive relationships. For each policy and state s, the following consistency condition applies between the value of s and the value of its possible subsequent states: This equation is also called the Bellman equation.

$$\begin{aligned} v_{\pi}(s) &= \mathbb{E}_{\pi}[R_t + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_{\pi}[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)[r + \gamma v_{\pi}(s')] \end{aligned}$$

For the Value Function, the Bellman equation defines a relation of the value of State s and its following State s'. The Bellman equation is also used for the Action-Value function. Accordingly, the Action-Value can be calculated from the following state:

In the Bellman equations, the structure of the MDP formulation is used to reduce this infinite sum to a system of linear equations. By directly solving the equation, the exact state values can then be determined.

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[R_t + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_a \pi(a|s) q_\pi(s', a')]
\end{aligned}
$$

3.4 Problem of Finding Optimal Policy

To solve a task or a problem in RL means to find a policy that will have a great reward in the long run. For finite MDPs, an optimal policy can be precisely defined in the following way. Value Functions define a partial order over different policies. For example, a policy π is better or at least as good as a policy π' if the expected return across all states is greater than or equal to that of π'. In other words, $\pi \geq \pi'$ is better if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all states s. This is an optimal policy π^*.

Although there may be several optimal policies, they all share the same state value function, which is called the optimal state value function and is defined as follows:

$$v_*(s) = \max_\pi v_\pi(s) \tag{4}$$

Optimal policies also share the same optimal action-value function:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{5}$$

Due to the fact that v_* is a value function for a policy, it must meet the condition of uniformity of the Bellman equation. Since it is the optimal value function, the consistency condition of v_* can be written in a special form without reference to a specific policy. This Bellman equation for v_* is also called the Optimal Bellman Equation and can also be written down for the optimal action-value function. Once v_*

$$
\begin{aligned}
v_*(s) &= \max_{a(s)} q_{\pi*}(s,a) \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]
\end{aligned}
$$

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

exists, it is very easy to derive an optimal policy. There will be one or more actions for each state s, where a maximum in the optimal Bellman equation is reached. Any policy that assigns a probability greater than zero to only these actions is an optimal policy.

Using v_*, the optimal expected long-term return is converted into a quantity that is immediately available for each state. A one-step predictive search thus yields the optimal long-term actions.

With q_*, on the other hand, the agent does not have to perform a one-step predictive search. For each state s, only one action has to be found, which maximizes $q_*(s,a)$.

4 Q-learning

4.1 Basic Q-learning(Single Agent)

In Q-learning[Watkins and Dayan, 1992], the agent learns an action-value function, or Q-function, given the value of taking a given action in a given state. Q-learning always selects the action that maximizes the sum of the immediate reward and the value of the immediate successor state.

The Q-function for the policy π:

$$Q(s,a) = r(s,a) + \gamma V^\pi(\delta(s,a)). \tag{6}$$

Therefore, the optimal policy π^* in terms of Q-function:

$$\pi^*(s) = \arg\max_a [Q(s,a)] \tag{7}$$

The update rule for the Q-function is defined as follows:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right), \tag{8}$$

which is calculated when action a is executed in state s leading to the state s'. Here $0 \leq \alpha \leq 1$ is a learning rate parameter and γ is the discount factor. (See, e.g., (Mitchell, 1997) for more details.)

4.2 Parameters of Q-learning

In Q-learning, the discount factor (γ) and the learning rate (α) play crucial roles in updating Q-values.

The discount factor, γ, determines the importance of future rewards in the agent's decision-making process. A value closer to 1 means the agent considers future rewards more heavily, while a value closer to 0 means it prioritizes immediate rewards.

The learning rate, α, controls the extent to which newly acquired information overrides old information. A high learning rate leads to rapid updates but may cause instability, while a low learning rate leads to slower but more stable learning.

4.3 Deep Q-learning(Single Agent)

Google DeepMind introduced deep Q-learning, merging Convolutional Neural Networks (CNNs) with basic Q-learning [Hester et al., 2018] [et al., 2013]. This method overcomes the challenges of expressing the value function for every state by employing a CNN for function approximation [Hester et al., 2018] . Deep Q-learning integrates experience replay and the target Q technique to stabilize the learning process.

Experience replay mitigates the instability caused by correlations between states, actions, and rewards by storing experiences in a buffer and randomly sampling them [36]. The target Q technique separates the target network and Q network to reduce correlations and improve stability [et al., 2013].

Deep Q-learning addresses the correlation issue by collecting diverse samples and randomly selecting them during training [Zhang et al., 2017]. However, excessive memory usage may slow down learning.

4.4 Epsilon-Greedy Strategy

Exploration vs Exploitation : Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future.

Exploitation on the other hand, chooses the greedy action to get the most reward by exploiting the agent's current action-value estimates. But by being greedy with respect to action-value estimates, may not actually get the most reward and lead to sub-optimal behaviour. When an agent explores, it gets more accurate estimates of action-values. And when it exploits, it might get more reward. It cannot, however, choose to do both simultaneously, which is also called the exploration-exploitation dilemma.

Epsilon-Greedy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly. The epsilon-greedy, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring.

4.5 Pseudo-code

Algorithm 1: Q-learning algorithm

Input: Initialize state-action function $Q(s, a)$

while *not converged* **do**

- Present current state S_t;
- Calculate optimal action;
- Execute selected action (ϵ-greedy) a_t;
- Observe new state and reinforcement signal, S_{t+1} and r_{t+1}, respectively;
- Update state-action function as: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$;
- New state becomes current state;

Source : [Mitić et al., 2011]

5 Maze Problem

5.1 Implementation of the Maze environment

In order to analyze hyperparameters for Q-learning, a Maze environment and a Q-learning algorithm had to be set up. The environment was constructed using OpenAI Gym [OpenAI, 2019], a toolkit for developing and comparing reinforcement learning algorithms, and Pygame [Shinners, 2011], a library for creating interactive applications and games in Python. While there are existing OpenAI Gym environments for mazes, we opted to create our own implementation from scratch to tailor it to our specific requirements.

The environment provides the agent with rewards when it performs actions. Three types of rewards were defined: passive, exit, and collision. The passive reward which is -1 is given when the agent moves without reaching the exit or colliding with a wall. The exit reward is awarded when the agent successfully reaches the exit which is 100, and the collision reward is given when the agent collides with a wall which is -10. A score counter is also implemented to keep track of the agent's performance. Figure 4 illustrates the maze environment.

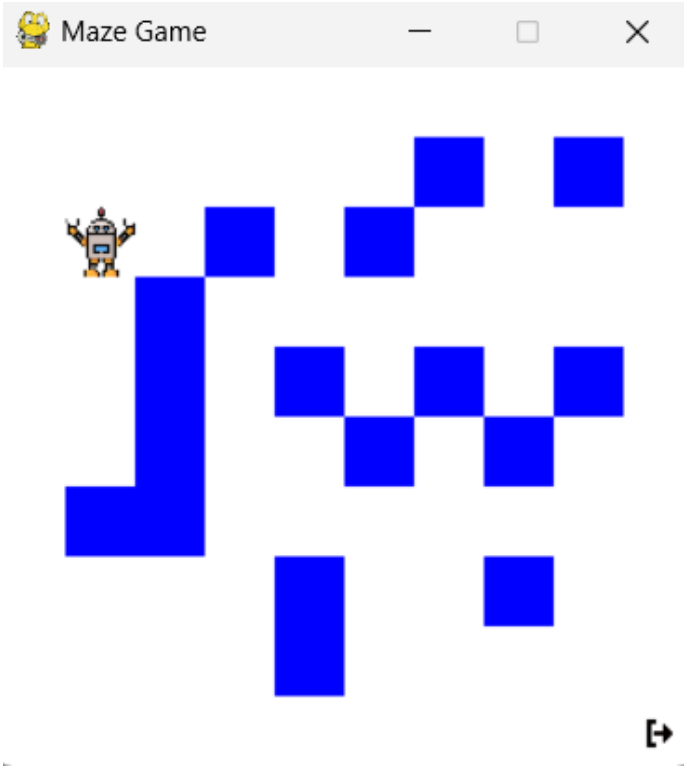

Figure 4: Illustration of the maze environment.

5.2 Number of walls

Determining the number of walls in the maze grid holds significant importance, particularly in shaping the complexity of the problem and facilitating its resolution. In our case, allocating precisely 20% of the total grid cells to walls plays a pivotal role in defining this aspect.

This aspect is crucial for ensuring an appropriate level of challenge and complexity in solving the maze navigation problem. The significance of setting the number of walls to 20% lies in balancing the maze's difficulty. Too few walls may result in a relatively straightforward path to the goal, limiting the learning experience and failing to fully exploit the capabilities of the reinforcement learning algorithm. On the other hand, an excessive number of walls could make the maze overly complex, potentially leading to challenges in finding an optimal solution within a reasonable timeframe.

Therefore, by allocating approximately one-fifth of the grid cells as walls, we strike a balance between creating a challenging environment for the agent to navigate and ensuring that the maze remains solvable within a reasonable computational effort. This approach enhances the effectiveness of the reinforcement learning algorithm by providing a suitable environment for learning optimal navigation strategies.

The formula for the density of walls is given by:

$$n_{\text{walls}} = \text{int}(0.2 \times (n_{\text{rows}} \times n_{\text{cols}}))$$

This formula represents 20% of the total number of cells in the grid, and the fixed density will be 20%.

5.3 Q-learning implementation

A simple function for training the agent was implemented using the update rule, equation 8 as its core. The function `step()` method of the `MazeEnv` class. The following code snippet illustrates the Q-learning process:

```
def step(self, action):
    row, col = self.robot_position
    if action == 0:  # move up
        row -= 1
    elif action == 1:  # move down
        row += 1
    elif action == 2:  # move left
        col -= 1
    elif action == 3:  # move right
        col += 1
    elif action == 4:  # move up-left
        row -= 1
        col -= 1
    elif action == 5:  # move up-right
        row -= 1
        col += 1
    elif action == 6:  # move down-left
        row += 1
        col -= 1
    elif action == 7:  # move down-right
        row += 1
        col += 1

    # Ensure the robot stays within the maze
    row = max(0, min(row, self.rows - 1))
    col = max(0, min(col, self.cols - 1))

    # Check if the new position is a wall
    if self.maze[row][col] == 1:
        reward = self.collision_penalty
    else:
        self.robot_position = (row, col)
        if self.robot_position == self.exit_position:
            reward = self.exit_reward  # Reached the exit
        else:
            reward = self.passive_reward  # Moved to a valid position

    # Update score
    self.score += reward

    # Update Q-table
    old_q_value = self.q_table[self.robot_position[0], self.robot_position[1], action]
    max_future_q = np.max(self.q_table[self.robot_position[0], self.robot_position[1]])
    new_q_value = (1 - self.alpha) * old_q_value + self.alpha * (reward + self.gamma * max_future_q)
    self.q_table[self.robot_position[0], self.robot_position[1], action] = new_q_value

    done = self.robot_position == self.exit_position
    return self.robot_position, reward, done, {'score': self.score}  # Return score in the info dict
```

In this code snippet, the `step()` method takes an `action` as input, representing the movement of the robot within the maze. It updates the robot's position, checks for collisions with walls, updates the score, and then applies the Q-learning update rule to the Q-table based on the received reward and the maximum expected future reward. Finally, it returns the new state (position), reward, whether the episode is done, and the score.

5.4 Hyperparameters

Hyperparameters play a crucial role in determining the performance of reinforcement learning algorithms. In our maze navigation task, we focus on four key hyperparameters: the learning rate (α), the discount factor (γ), the exploration rate (ϵ), and the grid size.

We define the default values and intervals for each hyperparameter as follows:

Table 1: Default values and intervals for hyperparameters

Hyperparameter	**Default Value**	**Interval**
Learning Rate (α)	0.1	[0.01, 0.1, 0.5, 0.9]
Discount Factor (γ)	0.9	[0.01, 0.5, 0.9, 0.99, 0.999, 0.9999]
Exploration Rate (ϵ)	0.3	[0.9,0.5,0.4,0.3]
Grid Size	(7, 7)	[(7, 7), (10, 7), (15, 10), (17, 12), (30, 30)]

These default values serve as initial settings for our experiments, while the specified intervals allow for exploration of a range of values during hyperparameter tuning.

we will compare the performance based on two metrics: the time taken to find a solution and the number of cycles taken to find the path to the exit. By systematically varying the hyperparameters within the specified intervals and measuring these performance metrics, we aim to identify the optimal hyperparameter values that result in the **fastest** solution time and the **fewest** number of **cycles**.

6 Results

6.1 Parameter tests

The results of our tests are displayed in the figures below. For each experiment, we varied one parameter at a time. In each figure, the y-axis represents the score(Time / Cycles), while the x-axis shows the variation of the parameter being tested. We created plots illustrating the score(Time / Cycles) for each parameter value. This approach allows us to visualize how changes in individual parameters affect both the score and the time taken for Q-learning to converge. The title of each graph says which hyperparameter being varied, accompanied with a textbox containing the different parameter values being tested. Other values remain at its default value shown in table 1.

6.1.1 Learning rate and Gamma

We present the results of varying hyperparameters directly related to the Q-learning update rule 8. Our experiments were conducted on a fixed grid size of 7x7.

We begin by exploring the impact of the **learning rate**. Figures 5 and 6 display the scores (Time / Cycles) corresponding to different learning rate values. The figures demonstrate comparable rates of convergence for learning rates of 0.1 and 0.01, accompanied by similar time durations. However, learning rates exceeding 0.5 result in significantly longer convergence times and require a higher number of cycles to find the optimal path.

We observe that a learning rate between 0.1 and 0.01 appears to be most optimal. Thus, we aim to pinpoint a value within this range.

For instance, when employing a learning rate of 0.05, the following results were obtained: Elapsed time: 18.067 seconds Number of cycles: 1205

Varying the **gamma** parameter (Figures 7 And 8)in our Q-learning algorithm showed that lower values led to slower

convergence and higher computational times. Moderate values around 0.5 resulted in faster convergence, while values closer to 1 increased convergence time significantly. Extreme values closer to 0 or 1 showed either slow convergence or very high computational costs.

As demonstrated in [Gite, 2017], when gamma is closer to zero, the agent tends to prioritize immediate rewards, whereas when gamma is closer to one, the agent assigns greater weight to future rewards, indicating a willingness to delay gratification.

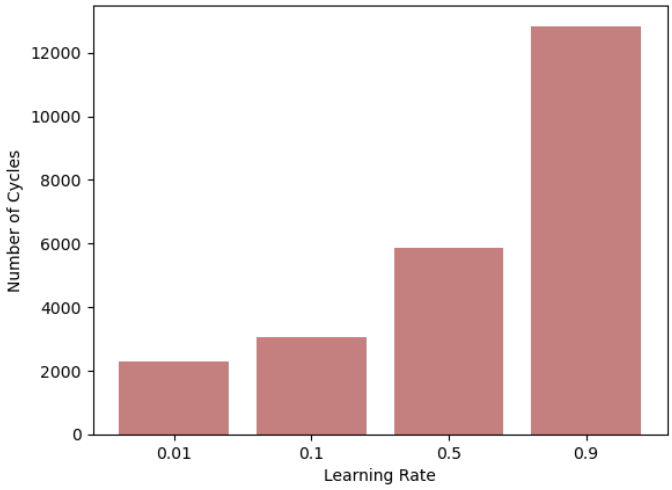

Figure 5: Number of Cycles for each Learning rate

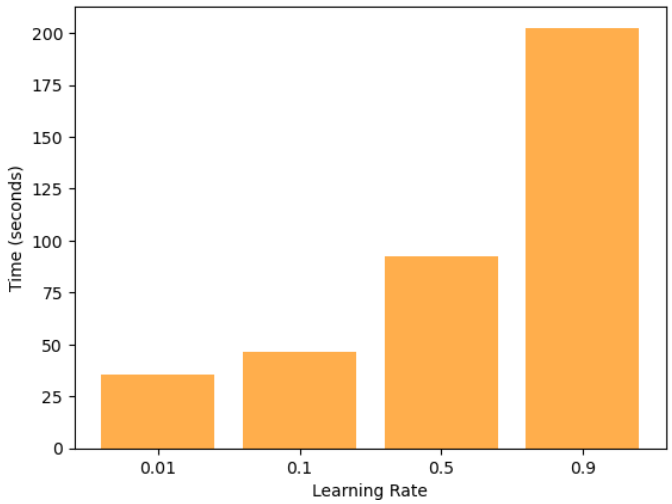

Figure 6: Time spent for each Learning rate

6.1.2 Epsilon : Exploration and exploitation

Lastly we investigate how the balance between exploration and exploitation affects the number of cycles and convergence time results. Firstly we begin by varying the minimum values for epsilon which results are shown in Figures 9 and 10. As we mentionned before, epsilon is a probability value where 1.0 is treated as 100 % chance for the agent to select a random action while 0.0 is 0 % chance.

When epsilon was set to 0.9, the agent exhibited relatively efficient behavior, completing the task in 378 cycles and approximately 8.44 seconds. A slight reduction in epsilon to 0.5 further improved performance, with 330 cycles and a convergence time of about 4.86 seconds. However, as epsilon decreased to 0.4 and 0.3, the agent's computational demands increased substantially. For epsilon equal to 0.4, the agent required 5737 cycles and approximately 86.07 seconds, while epsilon equal to 0.3 resulted in 11190 cycles and a convergence time of around 165.60 seconds.

These findings underscore the delicate balance between exploration and exploitation in reinforcement learning tasks. Higher epsilon values encourage more exploration, potentially leading to a more comprehensive understanding of the environment but may incur higher computational costs. Conversely, lower epsilon values favor exploitation, exploiting the agent's existing knowledge but risking premature convergence to suboptimal solutions.

In addressing this **trade-off**, decreasing epsilon over time through techniques like epsilon decay or annealing emerges as a valuable strategy. By gradually reducing epsilon as the agent gains more experience, it can effectively transition from exploration to exploitation, adapting its strategy to maximize learning efficiency and performance in maze navigation tasks. Therefore, integrating epsilon decay methods into the learning process is essential for optimizing the agent's exploration-exploitation balance and enhancing its overall learning effectiveness.

6.1.3 Grid size

Varying the grid size parameter is shown in Figures 11 and 12. The figures shows similar rates of convergence for the four grid sizes 7*7, 10*7 and 15*10 while the grid size 30*30 takes longer to converge. The variation in grid size significantly influences the computational demands and performance of the reinforcement learning algorithm. As the maze size increases, the complexity of the environment grows, leading to a proportional increase in the number of possible states and actions that the agent must explore. Consequently, larger mazes require more extensive exploration and computation to identify the optimal path to the goal state. This relationship between grid size and computational complexity underscores the importance of carefully considering maze size when designing and implementing reinforcement learning algorithms for maze navigation tasks. One thing to keep in mind is that larger grid sizes will increase the amount of Q-table updates in an episode which involves more computational work.

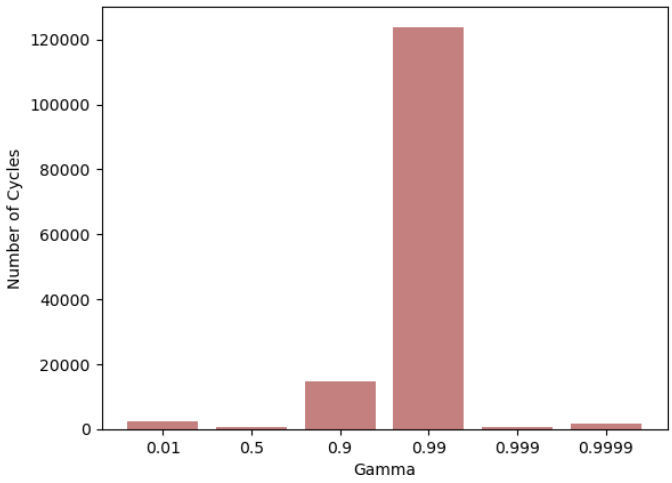

Figure 7: Number of Cycles for each Gamma

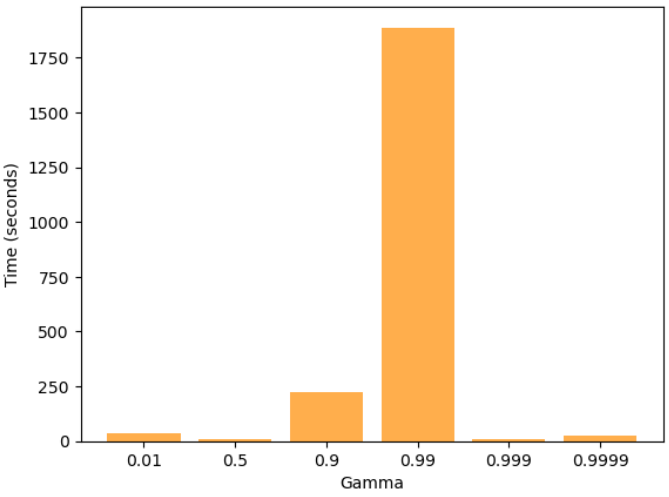

Figure 10: Time spent for each Gamma

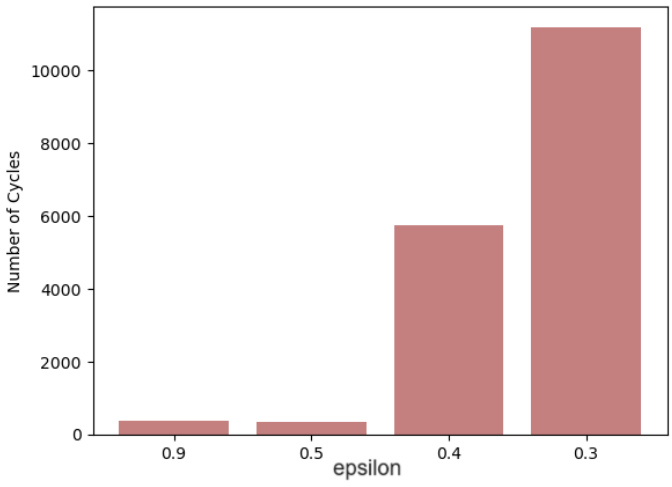

Figure 8: Number of Cycles for each Epsilon

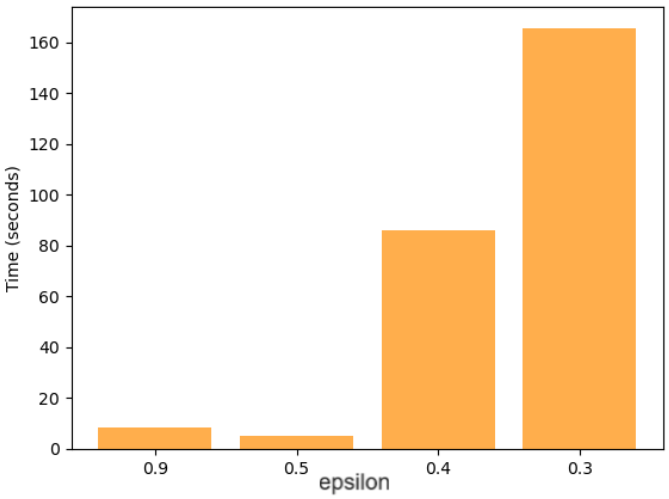

Figure 11: Time spent for each Epsilon

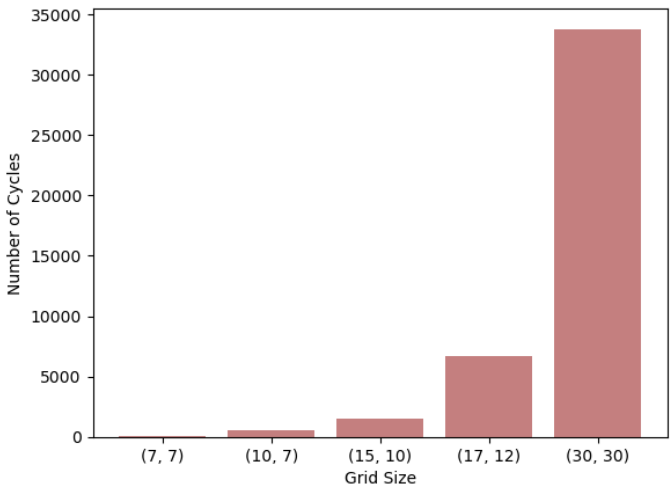

Figure 9: Number of Cycles for each Grid Size

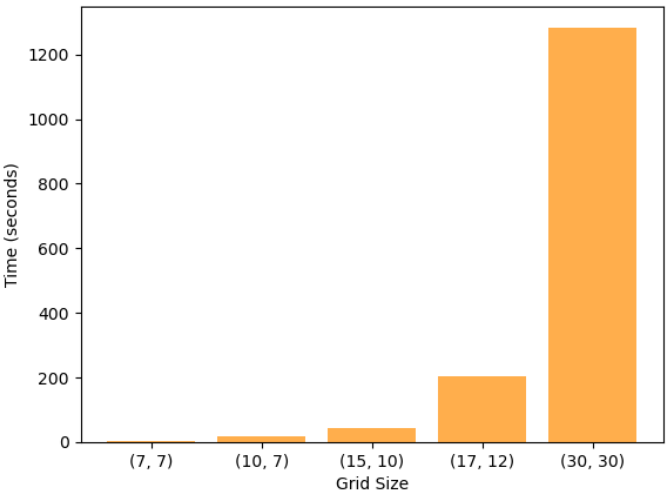

Figure 12: Time spent for each Grid Size

6.2 The best parameter setup

By using the results from section 6.1 we construct an optimal parameter setup with regards to fastest learning rate and highest achieved score. This resulted in the values shown in table 2.

Table 2: Comparison of the default parameters and the optimal setup.

Hyperparameter	Default Value	Optimal
Learning Rate (α)	0.1	0.05
Discount Factor (γ)	0.9	0.5
Exploration Rate (ϵ)	0.3	0.8
Grid Size	(7, 7)	(7, 7)

The comparison of Default and Optimal Hyperparameters for Maze Size (7, 7) is presented in Table 3. As shown in the Table and in the Figures 13 and 14, the optimal hyperparameter setup yields significantly better results compared to the default hyperparameters.

Table 3: Comparison of Default and Optimal Hyperparameters

	Default	Optimal
Cycles	19587	199
Time (seconds)	562.11	5.65

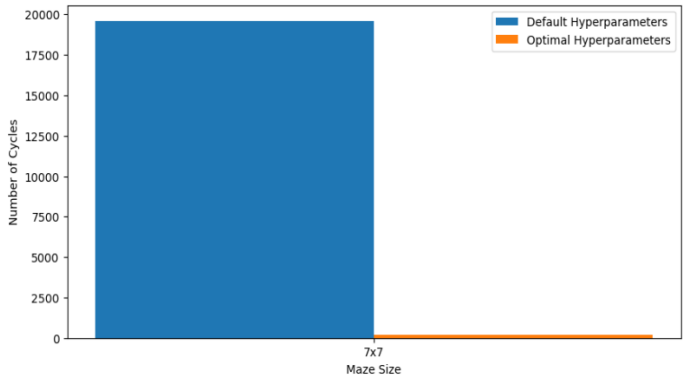

Figure 13: Number of Cycles (Default vs.Optimal)

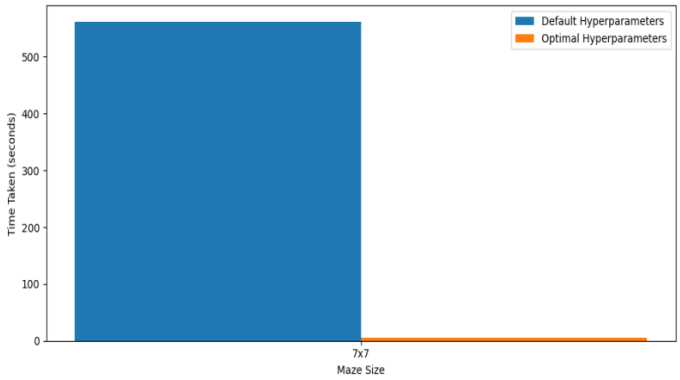

Figure 14: Time Spent (Default vs.Optimal)

7 Discussion

7.1 Grid Size

Our findings highlight how grid size affects maze navigation complexity. Larger mazes demand more exploration and computation due to their increased state and action space. Smaller grids, , such as 7x7, 10x7, and 15x10, show comparable convergence rates, but a 30x30 grid requires longer convergence times and higher computational resources. This emphasizes the need to balance efficiency with performance, especially in larger maze environments.

7.2 Learning Rate and Gamma

- **Learning Rate (α):** One notable variation we consider is the impact of different learning rates on algorithm performance. Our findings indicate that moderate learning rates between 0.1 and 0.01 result in faster convergence, striking a balance between exploring new paths and exploiting existing knowledge. However, excessively high or low learning rates lead to suboptimal performance, either prolonging convergence time or hindering exploration.
- **Discount Factor (γ):** The discount factor's impact on convergence and computational efficiency became evident through our experiments. Moderate discount factors around 0.5 demonstrate faster convergence, effectively balancing immediate rewards with future rewards. Extreme values near 0 or 1 result in slower convergence or increased computational costs, highlighting the importance of selecting an appropriate discount factor.

7.3 Epsilon: Exploration and Exploitation

The results underscored the delicate balance required to optimize performance: Higher exploration rates encourage the agent to explore new paths, potentially enhancing its understanding of the maze environment. However, this comes at the cost of increased computational resources. Lower exploration rates favor exploitation, leveraging existing knowledge to navigate the maze efficiently but risk premature convergence to suboptimal solutions.

In summary, our experiments shed light on the performance characteristics of the Q-learning algorithm in maze navigation tasks. While standard Q-learning demonstrates effectiveness in moderate maze sizes, variations enhanced by hyperparameters offer opportunities for optimization. These findings provide valuable insights for designing and implementing reinforcement learning algorithms in maze navigation and similar tasks.

Conclusion

The delicate balance between exploration and exploitation plays a crucial role in the efficiency of Q-learning algorithms. By carefully tuning these parameters, significant reductions in training time can be achieved when employing a Q-table approach. It's essential to note that optimal learning rate and gamma values, as identified in our study for Maze, may not universally apply to all Q-learning problems.

Future work

Future research avenues could explore dynamic approaches to hyperparameter tuning, such as adaptive learning rates or epsilon decay strategies. Additionally, investigating the generalizability of our findings across diverse reinforcement learning environments could further enrich our understanding of hyperparameter optimization in complex tasks.

References

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684.

[Dittert, 2020] Dittert, S. (2020). Reinforcement learning: Value function and policy. *Analytics Vidhya*.

[et al., 2013] et al., V. M. (2013). Playing atari with deep reinforcement learning. *arXiv*.

[Gite, 2017] Gite, S. (2017). Practical reinforcement learning — 02 getting started with q-learning. *Towards Data Science*. Published in Towards Data Science.

[Hester et al., 2018] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., Dulac-Arnold, G., Agapiou, J., Leibo, J., and Gruslys, A. (2018). Deep q-learning from demonstrations. In *Proc. 32nd AAAI Conf. Artif. Intell.*, pages 3223–3230.

[Markov, 1954] Markov, A. A. (1954). The theory of algorithms. *Trudy Mat. Inst. Steklov*, 42:3–375.

[Mitić et al., 2011] Mitić, M., Miljković, Z., and Babic, B. (2011). Empirical control system development for intelligent mobile robot based on the elements of the reinforcement machine learning and axiomatic design theory. *FME Transactions*, 39:1–8.

[OpenAI, 2019] OpenAI (2019). Openai gym. url: https://gym.openai.com/.

[Shinners, 2011] Shinners, P. (2011). Pygame - python game development. Retrieved from http://www.pygame.org.

[Sutton and Barto, 2015] Sutton, R. S. and Barto, A. G. (2015). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts; London, England, second edition, in progress edition.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292. Technical Note.

[Wikipedia, 2024] Wikipedia (2024). Markov decision process — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=1220124782. [Online; accessed 27-April-2024].

[Zhang et al., 2017] Zhang, Q., Lin, M., Yang, L. T., Chen, Z., and Li, P. (2017). Energy-efficient scheduling for real-time systems based on deep q-learning model. *IEEE Trans. Sustain. Comput.*, 4(1):132–141.