AI6101 Reinforcement Learning Assignment

Adnan Azmat (G2303265K)
MSc Artificial Intelligence, Nanyang Technological University

Abstract

This report presents the implementation and experimental evaluation of Q-learning, a fundamental reinforcement learning algorithm, to solve the CliffBoxPushing grid-world game. This study introduces two novel modifications to enhance the algorithm's performance. Firstly, a distance matrix is employed in place of the traditional Manhattan distance metric to measure the spatial relationship between the box and the goal. Secondly, various epsilon decay strategies are explored, particularly favoring early-stage exploration. The report provides a detailed account of the implementation process, the results of the experiments, as well as present few novel modifications to solve the CliffBoxPushing game.

Introduction

A. Grid-world Game

Table 1 below represents the grid-world environment. The dimensions of the grid is $6 * 14$. A represents the agent responsible for pushing the box (B) to the goal position (G). The x blocks are the cliff which the agent needs to avoid. In the table, the goal position is situated at $(3, 13)$, the agent is at $(5, 0)$ while the box is at $(4, 1)$.

						x	x						
						x	x						
			x			x	x					x	
			x			x					x	x	G
	B		x								x	x	
A			x								x	x	

Table 1: The Cliff Box Pushing Grid World

The game ends under three conditions:

1. The agent or the box steps into the dangerous region (cliff).
2. The current time step attains the maximum time step of the game.
3. The box arrives at the goal.

B. Markov Framework for Decision-Making

A Markov Decision Process (MDP) is a framework for decision-making in situations where actions at each step depend only on the immediate past action, not on prior events. MDPs provide a way to address decision-making under uncertainty in various real-world scenarios. In the next section the MDP formulation for the environment is described.

C. Environment Description

The environment is described by the following components:

- **State:** The state consists of the position of the agent and the box. In Python, it is represented as a tuple. For example, at time step 0, the state is represented as $((5, 0), (4, 1))$ where $(5, 0)$ is the position of the agent, and $(4, 1)$ is the position of the box.
- **Action:** The action space is defined as $[1, 2, 3, 4]$, which corresponds to $[\text{up}, \text{down}, \text{left}, \text{right}]$. The agent needs to select one of these actions to navigate in the environment.
- **Reward:** The reward is composed of several components:
  1. The agent will receive a reward of -1 at each time step.
  2. A negative value of the distance between the box and the goal.
  3. A negative value of the distance between the agent and the box.
  4. The agent will receive a reward of -1000 if the agent or the box falls into the cliff.
  5. The agent will receive a reward of 1000 if the box reaches the goal position.
- **Transition:** The agent's action can change its position and the position of the box. If a collision with a boundary happens, the agent or the box would stay in the same position. The transition can be seen in the `step()` function of the environment.

Q-Learning

Q-Learning is a model-free reinforcement learning algorithm. The primary goal of Q-Learning is to learn a policy that guides an agent on what actions to take in different states, ultimately maximizing the expected cumulative reward. One of the remarkable features of Q-Learning is its ability to handle problems with stochastic transitions and rewards without requiring a model of the environment.

For any finite Markov decision process (MDP), Q-Learning aims to find an optimal policy that maximizes the expected value of the total reward over successive steps, starting from the current state. The 'Q' in Q-Learning stands for the function that computes the maximum expected rewards for an action taken in a given state.

Q-Learning employs a recursive equation to compute Q, which relates to a state (s) and an action (a) as follows:

$$Q(s,a) = (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s',a'))$$

- s' is the new state resulting from taking action a.
- r is the reward received after taking action a at state s.
- α is the learning rate ($0 \leq \alpha \leq 1$), which determines the weight of new information compared to old information.
- γ is the discount factor ($0 \leq \gamma \leq 1$), which represents the importance of future rewards compared to immediate rewards.

The key steps in Q-Learning are:

1. **Initialization:** Start by initializing the Q-values for all state-action pairs, usually arbitrarily. For terminal states, set Q-values to 0. These Q-values will be updated as the agent explores and learns.
2. **Episodic Loop:** Q-Learning operates in episodes. In each episode, the agent interacts with the environment.
   (a) **State Initialization:** Begin by initializing the agent's starting state, denoted as S.
   (b) **Step Loop:** Now, the agent performs steps within the episode.
      i. **Action Selection:** Choose an action A from state S using a policy derived from the current Q-values. This policy could be, for example, epsilon-greedy, allowing for exploration and exploitation.
      ii. **Interaction with Environment:** Execute action A in the current state S and observe the immediate reward R and the resulting new state S'.
      iii. **Q-Value Update:** Update the Q-value for the current state-action pair $Q(S,A)$ using the Q-Learning equation. This incorporates the learning rate (α), the immediate reward (R), and the maximum expected future reward ($\gamma \cdot \max_{a'} Q(S',a')$).
      iv. **State Transition:** Update the current state to the new state, $S = S'$.
   (c) **Termination Check:** Continue the step loop until the current state becomes a terminal state, signifying the end of the episode.
3. **Repeat:** Repeat this episodic process for as many episodes as needed, allowing the agent to learn and improve its Q-values and, consequently, its policy.

The Q-Learning algorithm pseudo code can be written as:

Initialize $Q(s,a)$, for all s in S, a in $A(s)$, arbitrarily, and $Q(\text{terminal-state},.) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., epsilon-greedy)
    Take action A, observe R, S'
    Update $Q(S,A)$ as $Q(S,A) = Q(S,A) + \alpha \cdot [R + \gamma \cdot \max_a Q(S',a) - Q(S,A)]$
    $S = S'$
  until S is terminal

This process continues iteratively until the Q-values converge, indicating that the agent has learned an optimal policy.

Modifications

A. Distance Matrix

The initial reward calculation formula uses the Manhattan distance when calculating the distance between points, as given by:

$$\text{dist} = \sum |\text{self.agent\_pos} - \text{self.box\_pos}|$$

However, in the grid world containing cliffs, the box and agents cannot traverse through them. The Manhattan distance does not take into the consideration the layout of the environment. To address this issue, I propose calculating a distance matrix where each element $D(i,j)$ represents the actual shortest possible distance to the goal G. We assign a large distance penalty of 1000 to the cliffs, and reaching the destination is rewarded with a distance of -1000. So for the given grid-box environment, we can compute the matrix as follows:

						x	x						
						x	x						
			x			x	x					x	
			x			x					x	x	G
	B		x								x	x	
A			x								x	x	

Table 2: The Cliff Box Pushing Grid World

17	16	15	14	13	12	1000	1000	9	8	7	6	5	4
16	15	14	13	12	11	1000	1000	8	7	6	5	4	3
17	16	15	1000	11	10	1000	1000	7	6	5	4	1000	2
18	17	16	1000	10	9	1000	7	6	5	4	3	2	1
19	18	17	1000	9	8	7	6	5	4	3	2	1	-1000
20	19	18	1000	10	9	8	7	6	5	4	1000	1000	1

Table 3: Distance matrix (cliffs = 1000 | goal = -1000)

The distance matrix with penalties for cliffs and rewards for goal achievement promotes safe exploration by discouraging dangerous actions, encourages efficient navigation by considering obstacles, and provides a clear reward structure that accelerates learning and results in robust, adaptable policies. Hence, we can write the distance between the box and the agent using the distance matrix itself as:

$$\text{dist} = |D(x_a, y_a) - D(x_b, y_b)|$$

With the use of the distance matrix, we can now simplify the reward system by:

- Eliminating the reward condition: The agent will no longer receive a reward of -1000 if the agent or the box falls into the cliff.
- Eliminating the reward condition: The agent will no longer receive a reward of 1000 if the box reaches the goal position, as the distance matrix now handles these cases.

Hence the **new efficient reward system** is the sum of:

1. The agent will receive a reward of -1 at each time step.
2. The negative value of the distance between the box and the goal using the distance matrix: $(D(x_b, y_b))$.
3. The negative value of the distance between the agent and the box using the distance matrix: $|D(x_a, y_a) - D(x_b, y_b)|$.

B. Epsilon Decay

In reinforcement learning, epsilon decay is a technique used to balance exploration and exploitation. At the start of training, the agent has a high epsilon value, which encourages random actions for exploration. As training progresses, epsilon decays, reducing the probability of random actions and increasing the likelihood of actions based on learned Q-values. This shift from exploration to exploitation allows the agent to leverage its learned knowledge. The decay rate is typically set such that epsilon approaches a small constant, ensuring some level of continued exploration.

Exponential Decay: This method reduces ϵ exponentially over time, which allows for a lot of exploration in the early stages of learning and gradually shifts towards exploitation as the agent learns more about the environment. The mathematical representation of exponential decay is:

$$\epsilon = \epsilon_{\text{min}} + (\epsilon_{\text{max}} - \epsilon_{\text{min}}) \cdot e^{-\text{decay rate} \cdot \text{episode}}$$

where ϵ_{max} and ϵ_{min} are the maximum and minimum epsilon values, respectively, and 'episode' represents the current episode number.

Linear Decay: In contrast, linear decay reduces ϵ at a constant rate. This provides a steady shift from exploration to exploitation throughout the learning process. The mathematical representation of linear decay is:

$$\epsilon = \epsilon_{\text{max}} - (\text{decay rate} \cdot \text{episode})$$

Experiment Results

In this experiment, I employed an exponential epsilon decay strategy in conjunction with the distance matrix approach (discussed earlier), where ϵ_{start} is initialized at 0.5 to encourage initial exploration, $\epsilon_{\text{decay\_constant}}$ of 0.001 ensures a gradual exploration reduction, ϵ_{min} at 0.1 guarantees some ongoing exploration, and you balance learning with a low α (0.1) and prioritize future rewards with a high γ (0.99). This configuration enables your agent to explore, learn, and make decisions efficiently in its environment.

A. Effect of parameters

In this section we explored the effects of different hyperparameters on the rewards vs episodes graph.

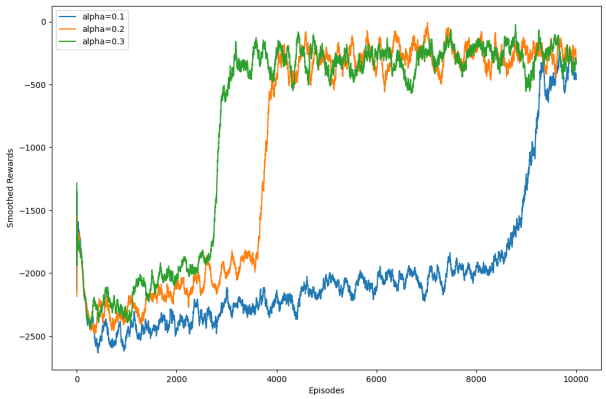

Figure 1: Rewards vs Episodes: Different values of alpha

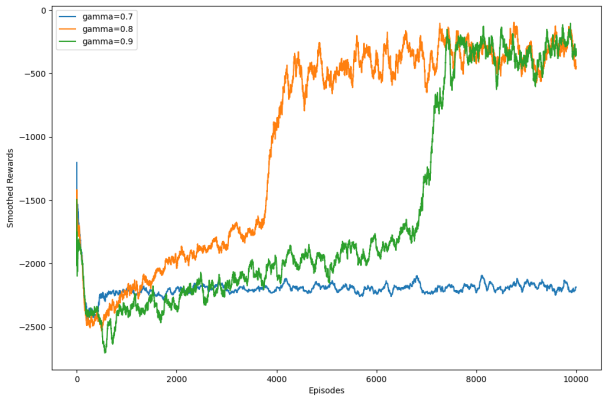

Figure 2: Rewards vs Episodes: Different values of gamma

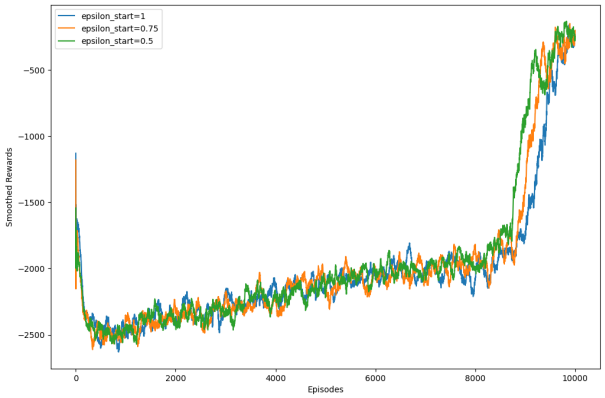

Figure 3: Rewards vs Episodes: Different values of epsilon_start

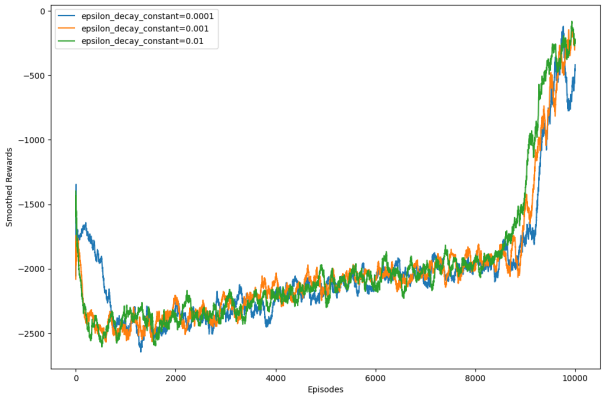

Figure 4: Rewards vs Episodes: Different values of epsilon_decay_constant

We use the following parameters for our experiment:

Variable	Value
ϵ_{start}	0.5
$\epsilon_{\text{decay\_constant}}$	0.001
ϵ_{min}	0.1
α (learning rate)	0.1
γ (discount factor)	0.99

Table 4: Parameters for the experiment

B. Rewards vs Episodes

This graph shows the relationship between rewards and episodes during the reinforcement learning experiment. It illustrates how the cumulative rewards change over time as the agent learns and interacts with its environment. The x-axis represents the number of episodes or iterations of training, while the y-axis represents the cumulative rewards earned by the agent.

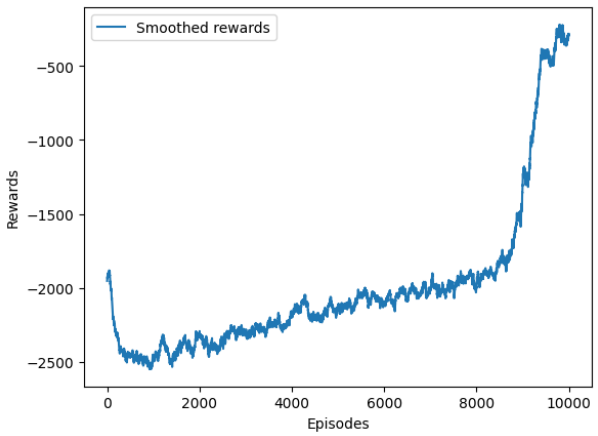

Figure 5: Smoothed rewards vs episodes

C. V-Table heatmap

Value Table is a data structure used to store the estimated values of states in a Markov Decision Process (MDP). These state values represent the expected cumulative reward the RL agent can achieve when starting in a specific state and following a particular policy. The V-table helps the agent make informed decisions by assessing the values of states and selecting actions that maximize expected cumulative rewards.

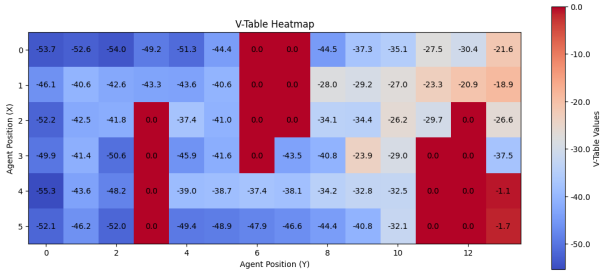

Figure 6: V-Table Heatmap

D. Episode-Run

In our experiments, the agent from position (5,0) was being to reach the goal at (4,13) in 37 steps. We take the action sequences the agent takes and print it to visualise the path the agent takes during the reward.

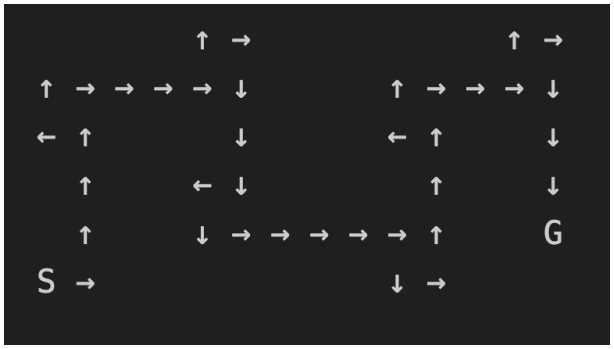

Figure 7: Path-taken by the agent during an episode run

```
step: 36, state: (2, 13, 3, 13), actions: 2, reward: -3
Action: 2
[[b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'_' b'_']
 [b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'_' b'_']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'x' b'A']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'x' b'x' b'B']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'G']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_']]
step: 37, state: (3, 13, 4, 13), actions: 2, reward: 998
Action: 2
[[b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'_' b'_']
 [b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'_' b'_']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'x' b'x' b'_' b'_' b'_' b'_' b'x' b'_']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'x' b'x' b'A']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'B']
 [b'_' b'_' b'_' b'x' b'_' b'_' b'_' b'_' b'_' b'_' b'_' b'x' b'x' b'_']]
rewards: 480
```

Figure 8: Grid layout during the final steps

Conclusion

In this report, we explored the implementation and experimental evaluation of Q-learning, a fundamental reinforcement learning algorithm, in the context of solving the CliffBoxPushing grid-world game. The objective was to develop an effective strategy for the agent to safely navigate the environment, avoid cliffs, and reach the goal efficiently. Two novel modifications were introduced to enhance the algorithm's performance: the use of a distance matrix to measure spatial relationships and various epsilon decay strategies to balance exploration and exploitation.

The introduction of a distance matrix was a key enhancement to the algorithm. This matrix allowed for a more realistic evaluation of distances in the grid world, considering the presence of cliffs that the agent and the box cannot traverse through. The distance matrix provided a clear and realistic representation of the shortest possible distances to the goal, assigning a high penalty to cliffs and a negative distance (high reward) to the goal position. This approach led to safer exploration by discouraging dangerous actions, promoting efficient navigation by considering obstacles, and simplifying the reward system. The new reward system took into account the negative values of the distance between the box and the goal as well as the distance between the agent and the box using the distance matrix. This allowed the agent to learn more effectively and make informed decisions.

The second modification involved exploring various epsilon decay strategies to balance the agent's exploration and exploitation. The exponential epsilon decay strategy was chosen, allowing for a high level of exploration in the early stages of learning and gradually shifting towards exploitation as the agent gained more knowledge about the environment. The epsilon decay strategy was a critical component of the agent's learning process, ensuring that it continued to explore the environment while also leveraging its learned Q-values to make better decisions.

The experimental results demonstrated the effectiveness of these modifications. The smoothed rewards versus episodes graph illustrated how the cumulative rewards changed over time as the agent learned and interacted with the environment. The agent's rewards showed a clear upward trend, indicating that it was learning and achieving higher rewards as it explored the environment. The V-Table heatmap provided a visual representation of the estimated values of states, helping the agent make informed decisions. Finally, an episode run showed a specific path taken by the agent during an episode, highlighting its ability to navigate the environment safely and reach the goal efficiently.

In conclusion, the implementation of Q-learning with the introduction of a distance matrix and epsilon decay strategies proved to be effective in solving the CliffBoxPushing grid-world game. These modifications enhanced the agent's ability to explore, learn, and make decisions efficiently. The agent's performance improved over time, and it demonstrated the capacity to find a safe path to the goal while avoiding cliffs. These modifications could be applied to other grid-world and navigation problems, making Q-learning a valuable tool for reinforcement learning tasks.