Reinforcement Learning

Arthur Yu, Afiq Ahmad Nordin, Dovudkhon Abdubokiev

Level: 3rd year, Credit Points: 20cp
Supervisor: Dr.Vladislav Tadic
Date of Submission: 14 November 2024

Acknowledgement of Sources

For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report. All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced. Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page or book, article etc.

Signed: Dovudkhon Abdubokiev, Afiq Ahmad Nordin, Arthur Yu
Date: 14 November 2024

Contents

Abstract

Reinforcement learning is an important machine learning paradigm that allows agents to learn through trial and error and improve its decisions based on feedback from its actions, like humans. In this report, we explored several key concepts in reinforcement learning. We began with an introduction to reinforcement learning as a whole, then focused on Discounted Reward Markov Decision Processes (MDPs) to establish the foundational framework. Then, we explored Temporal Difference (TD) Learning, which is probably the most central idea to reinforcement learning. These explorations were followed by introduction to two most popular TD-control methods, SARSA and Q-Learning.

1 Introduction to Reinforcement Learning

Reinforcement learning is a class of approximation methods for Markov Decision Processes(MDPs) based on optimization and Monte Carlo sampling. In simpler terms, reinforcement learning is goal-directed learning through interaction. The learner uses trial and error to discover which actions yield the most rewards. Reinforcement learning has two main features: trial-and-error search and delayed reward. It learns without labels and relies on feedback in the form of rewards or penalties.

Reinforcement learning is different from supervised learning. In supervised learning, a set of labelled examples is provided by a more knowledgeable supervisor for training. The training data is paired with labels that describe the action needed to be taken when certain data is fed in, allowing the system to categorise the data. This helps it respond correctly to data that was not included in the training set.

Reinforcement learning is also distinct from unsupervised learning, where a structure is found within an unlabelled set of data. Although, by definition, reinforcement learning and unsupervised learning share similarities, reinforcement learning focuses on maximising the reward rather than searching for hidden structures within the unlabelled data.

An example of reinforcement learning is an autonomous vehicle that learns about its environment by interacting with it. The vehicle takes actions, such as steering or accelerating, and observes whether these actions help it achieve its objective of reaching a specific destination. In reinforcement learning terms, this objective is associated with a reward when achieved, while actions that do not contribute to progress may result in no reward.

2 Understanding Discounted Reward MDPs

2.1 Definition of MDP and Markov Property

A Markov decision process (MDP) is a sequential decision problem where the underlying stochastic system evolves in a Markovian manner. In simple terms, an MDP models how an agent interacts with an environment to achieve a goal through decision-making over time. While MDPs provide the mathematical framework for reinforcement learning, they specify exact transition probabilities, allowing for precise theoretical analysis. MDPs model decision-making where outcomes are partly determined by the agent's actions and partly by randomness in the environment.

In a discounted reward MDP, which is often used in infinite-horizon set-

tings, the goal is to manage long-term decision-making under uncertainty. The goal of this type of MDP is to obtain a policy π that maximises the expected sum of discounted rewards over time under uncertainty. The discount factor, $\gamma \in (0, 1)$, scales future rewards, giving more weight to immediate rewards and diminishing the value of rewards that are farther in the future.

The Markov Property of a stochastic process refers to the process being "memoryless." This means the future state depends conditionally on only the current state and action, with the past history of states being ignored completely. Formally, it is written as:

$$P(S_{t+1} = s' | S_t = s) = P(S_{t+1} = s' | S_0 = s_0, S_1 = s_1, \ldots, S_t = s).$$

This property is important because it simplifies decision-making, as only the current state needs to be considered, making the process more efficient. In the upcoming section, the key components of MDP, including the states, actions, rewards, and transition function, will be thoroughly explored.

2.2 MDP Components: States, Actions, Rewards, Transition Function

In MDPs, as previously discussed, the interaction involves learning from a problem to achieve a certain objective. This interaction occurs between the agent and the environment. The agent is responsible for learning and making decisions, while the environment consists of everything outside the agent with which it interacts.

In discounted MDPs, the interaction between the agent and the environment is described as $M = (\mathcal{S}, \mathcal{A}, p, r, \gamma)$. The equation provided consists of states, actions, a transition function, and rewards. The detailed explanation of each component is as follows:

- **States**: $\mathcal{S}$ represents the state-space and denotes the set of all states. s represents the condition of the system in numerical terms. It is assumed to be finite or countably infinite for mathematical convenience.

- **Actions**: $\mathcal{A}$ represents the action-space and denotes the set of all available actions. Actions directly influence the system's behavior, and for mathematical convenience, it is often assumed to be finite. In this project, we will only consider MDPs with a finite state space.

- **Rewards**: r represents the reward function, which denotes the immediate reward received when the agent takes action a in state s.

- **Transition Function**: The transition function, $p(s', r|s, a)$, describes the probability of transitioning to state s' and receiving reward r given that action a is taken in state s. It defines how the system evolves based on the agent's actions.

The other component of M includes $\gamma \in [0, 1)$ as the discount factor, which scales future rewards, where $\Delta(\mathcal{S})$ denotes all possible transition probabilities over $\mathcal{S}$.

The interaction between the agent and the environment occurs in a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives information about the current state S_t and selects an action A_t.

A numerical reward, $R_{t+1} = r(S_t, A_t)$, for this action is received at the next time step. This action influences the new state S_{t+1} of the environment. The sequence of interactions continues as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

For a particular state $s' \in \mathcal{S}$, the transition function represents the probability of the next state s' occurring at time step t, given the previous state and action, as described by the following equation:

$$\begin{aligned} &P(S_{t+1} = s_{t+1}, R_{t+1} = r(S_t, A_t)|S_0 = s_0, A_0 = a_0, \ldots, S_t = s_t, A_t = a_t) \\ &= P(S_{t+1} = s_{t+1}, R_{t+1} = r(S_t, A_t)|S_t = s_t, A_t = a_t) = p(s_{t+1}, r|s_t, a_t). \end{aligned}$$

The following holds:

- $t \in \mathcal{T}$ represents any decision epoch.
- $s_0, \ldots, s_t, s_{t+1} \in \mathcal{S}$ is an arbitrary (deterministic) sequence of states.
- $a_0, \ldots, a_t \in \mathcal{A}$ is an arbitrary (deterministic) sequence of actions.
- $p(s_{t+1}, r|s_t, a_t)$ represents the probability of the system transitioning to state s_{t+1} and receiving reward r at time $t+1$, given that the system is in state s_t and action a_t is taken. This follows the equation stated above.

Since the next state s' depends only on the current state and action, and not on the previous history of states and actions, this system follows the Markov Property. In this report, we consider only the case where transitional probability p does not depend on time t.

Example 2.1 (Navigation)**.** To better understand the concept, let's apply it to a navigation problem. The current state represents the agent's current location. The available actions involve moving one step in one of four directions: north, south, east, or west. Assuming the step size is standardised, the transition function defines how the agent moves from the current location to a new location based on the selected action. The objective is to reach a specific target location. The agent receives a reward of 1 upon reaching the target and 0 otherwise. The discount factor $\gamma < 1$ encourages the agent to reach the goal state through the shortest path, by giving more weight to immediate rewards.

2.3 Value Function and Optimal Value Function

In reinforcement learning, the agent interacts with the environment to achieve a goal by maximising the cumulative rewards over time. The return, $G_{t,T}$, represents the cumulative future rewards starting from time step t. For simplicity, it is often defined as the sum of rewards:

$$G_{t,T} = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

where T is the final time step in an episode.

In order to take into account the time-importance of rewards, we introduce the discount factor, γ, i.e we replace the previous $G_{t,T}$ with the following:

$$G_{t,T} = \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t+1} R_T.$$

The importance of rewards decrease geometrically over time due to γ.

In the discounted MDP with *continuing episodic tasks*, return value satisfies the following recursive equation:

$$\begin{aligned} G_{t,\infty} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1,T} \end{aligned} \tag{2.1}$$

This is an important result that will serve as a foundation for the coming, more complex results.

The **policy**, π, is a mapping from states to probabilities of selecting each possible action. Many reinforcement learning algorithms focus on estimating value functions, which measure how good it is for the agent to be in a particular state based on the expected future rewards. The policy π determines the expected future rewards, which in turn influences the value functions.

We will now explore the different types of value functions and optimal value functions in detail.

2.3.1 State-Value Function (State One)

The state-value function, denoted as $v_\pi(s)$, is the expected return for the agent starting at state s while applying the policy π at any time step t. Formally, the state-value function is written as:

$$\begin{aligned} v_\pi(s) = \lim_{T\to\infty} \mathbb{E}_\pi[G_{t,T} \mid S_t = s] = \lim_{T\to\infty} \mathbb{E}_\pi \left[\sum_{k=0}^{T-1} \gamma^k R_{t+k+1} \mid S_t = s\right] \\ = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]. \end{aligned}$$

The return $G_{t,T}$ represents the cumulative future rewards, while γ is the discount factor that scales the future rewards.

2.3.2 Action-Value Function (State-Action One)

The action-value function, denoted as $q_\pi(s, a)$, is similar to the state-value function in terms of the expected return for the agent in state s, following a policy π. However, it also includes the action taken, a, in state s. We can formally define it as:

$$\begin{aligned} q_\pi(s,a) = \mathbb{E}_\pi[G_{t,T} \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \\ = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right]. \end{aligned}$$

2.3.3 Optimal State-Value Function (Optimal State One)

An optimal policy π_* is one where the expected return $v_\pi(s)$ is greater than or equal to the expected return under any other policy π. π_* is an optimal policy if and only if $v_{\pi_*}(s) \geq v_\pi(s)$ for all states s and all t.

The optimal state-value function is formally defined as:

$$v_*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S}.$$

This function gives the maximum expected return that can be achieved from any state s.

2.3.4 Optimal Action-Value Function (Optimal State-Action One)

The optimal action-value function follows the same optimality principle, but also accounts for the action taken by the agent, as in the action-value function. Formally, it is defined as:

$$q_*(s, a) = \max_\pi q_\pi(s, a).$$

Alternatively, it can also be written as:

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right].$$

This equation reflects both the return and the expected optimal value from the next state.

2.4 Bellman Equations for Policies

In previous subsections, we learned about return $G_{t,\infty}$ the cumulative future rewards starting from time step t and that it satisfies a recursive way as in Equation (2.1).

We also learned about state-value and action-value functions. Like in the case of returns, value functions also satisfy the similar recursive equations, which is a fundamental property for reinforcement learning. The **Bellman equations**, given below, provides a recursive decomposition of these values.

For state-value functions, we have:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_{t,T} \mid S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1,T} \mid S_t = s\right] \\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma \mathbb{E}_\pi\left[G_{t+1,T} \mid S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s')\right], \quad \text{for all } s \in \mathcal{S},
\end{aligned}$$

where $v_\pi(s)$ is the value of the current state, $\pi(a|s)$ represents the probability of taking action as in state s according to the policy π, $p(s', r|s, a)$ is the probability that the next state is s' and the agent receives reward r given

that the agent takes action a in the current state s, γ is the discount factor and $v_\pi(s')$ is the value of the next state.

For action-value functions, we have:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right], \end{aligned}$$

where $q_\pi(s,a)$ is the value of the state action pair (s,a).

2.5 Bellman Optimality Equations

An optimal policy, π_*, which is present in MDPs, is a policy that is stationary and deterministic. Both the state-value function and action-value function conditions are satisfied by the optimal policy π_*. The conditions are as follows:

$$v_{\pi_*}(s) = v_*(s), \quad q_{\pi_*}(s,a) = q_*(s,a).$$

The Bellman optimality equation is used to determine the optimal policy π_*. It produces the best possible action from the state by ensuring that the value of a state is equal to the maximum expected return.

The following are the Bellman optimality equation for $v_*(s)$, representing the state-value:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')]. \end{aligned}$$

Here, in the Bellman optimality equation, $p(s', r \mid s, a)$ is the following conditional probability:

$$p(s', r \mid s, a) = P(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$$

Similarly, for the action-value optimality equation:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right], \end{aligned}$$

where $p(s', r \mid s, a)$ is the probability of transitioning to state s' and a reward r is received given that the system is currently in state s and action a is taken.

If there are n states, the Bellman optimality equation is a system of n equations, in n unknowns in an environment, which the unknowns are the values of $v_*(s)$. When a system can be used to solve a system of non-linear equations for $v_*(s)$, it can be done the same for $q_*(s, a)$. When the equations are solved, then the step to find the optimal policy is taken by the Bellman equations stated before.

While $v_*(s)$ is used to find the optimal state-value, it can be seen as figuring out the short-term consequences of an action by looking one step ahead. With that, $q_*(s, a)$ is more optimal in taking into account the optimal actions, as actions are explicitly incorporated into the equations.

2.6 Policy Iteration and Value Iteration

Dynamic programming (DP) is a collection of algorithms to compute optimal policies for an MDP given that it has a perfect environment where the transition probabilities and rewards are fully known. DP generally uses value functions obtained to find the optimal policies in order to maximise the cumulative reward. Bellman optimality equations are employed to search for the optimal value functions that lead to the optimal policies.

For solving MDPs, two main methods are used: policy iteration and value iteration. Both are methods that alternate between evaluation and improvement to converge on an optimal solution.

2.6.1 Policy Iteration

Policy evaluation, also known as the prediction problem, is the ability to compute the state-value function v_π for an arbitrary policy π. It determines how good a policy π is by calculating the expected return from a state by following policy π. For updating the rule by iteration, policy evaluation uses the Bellman equation for v_π:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s \right] \\ &= \sum_{s', r} p(s', r \mid s, \pi(s))[r + \gamma v_k(s')]. \end{aligned}$$

The equation maps the states, $s \in S$, to $\mathbb{R}$ (real numbers). Iteratively, $\{v_k\}$ will converge to v_π as $k \to \infty$ by repeatedly applying the Bellman update equation to refine the value estimates for each state. This guarantees the existence and uniqueness of v_π under the MDP properties when $\gamma < 1$.

On the other hand, policy improvement is the process of searching for a new policy, π', that can improve the original policy to achieve a higher expected reward for each state by being greedy with respect to the value function of the current policy. This approach leads the process closer to an optimal policy. To determine which action maximises the return in each state, the action-value function $q_\pi(s, a)$ is used:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]. \end{aligned}$$

As the best action for each state under the initial policy is identified, a new, improved policy π' is created. It satisfies the policy improvement theorem:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad \Rightarrow \quad v_{\pi'}(s) \geq v_\pi(s).$$

The new policy, π', must be better than or at least equal to the previous policy, π. The new greedy policy is given by:

$$\pi'(s) = \arg\max_a q_\pi(s, a).$$

This update considers changes at all states and possible actions according to $q_\pi(s, a)$. The policy improvement theorem guarantees that this greedy approach satisfies the conditions for improving the policy.

Once a policy π has been improved to another policy π' using the value function v_π, a further improved policy π'' can be obtained through the evaluation and improvement process of $v_{\pi'}$. This sequence of improving policies and value functions is repeated until it converges to an optimal policy and an optimal value function. The process is guaranteed to converge in finite iterations because MDPs have a finite number of possible policies.

This iterative process is called policy iteration and can be visualised as follows:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

where each $\xrightarrow{E}$ denotes a policy evaluation, and each $\xrightarrow{I}$ denotes a policy improvement. Each policy is strictly better than the previous one in expected returns. Thus, policy iteration is a process involving both policy evaluation and policy improvement, repeated until reaching optimality.

2.6.2 Value Iteration

Value iteration is the process of combining policy evaluation and policy improvement in a single step. The equation for value iteration is derived from the Bellman equation and applied as an update rule as follows:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_k(s') \right]$$
$$= \max_a \mathbb{E} \left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a \right]$$

Value iteration is similar to policy evaluation but involves taking the maximum return over all actions taken. This equation will converge to v_* as $k \to \infty$, but in practice, the iteration stops once the difference between successive values is minimal. Value iteration prevents the inefficiencies of separate evaluation and improvement steps that can occur in policy iteration.

3 TD-Learning

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be Temporal Difference (TD)Learning". This is the description used in [5] to emphasize how important TD learning is. TD learning can be described as a combination of two other popular reinforcement learning ideas, Monte Carlo and Dynamic Programming. Like Monte Carlo methods, TD methods can learn from raw experience without needing the full knowledge of the environment transition probabilities. Like DP methods, TD methods update the estimates based on other estimates without needing to finish the whole episode. In summary, TD learning takes strong advantages of both these methods and combines them. We will use these methods for comparison often in the following subsections. Full details of Monte Carlo and DP methods are omitted in this report, but one can refer to [5] to get a deeper insight into these concepts.

In this section, we will explore Temporal Difference Learning, which is likely the most core concept in Reinforcement Learning. We will start by focusing on the prediction problem, then we will have a look at TD(0) algorithm. Lastly, we talk about the efficiency and convergence guarantee of TD-Learning.

3.1 TD Prediction

Like Monte Carlo, TD learning is a model-free learning algorithm, which means it uses samples rather than the distribution of the environment states

to make actions and come up with estimates. To understand better the difference between MC and TD methods we first look at the update rule for *constant-α* Monte Carlo method:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_{t,T} - V(S_t)].$$

where $G_{t,T}$ is the real value of return following time t, and α is a constant step-size parameter. The biggest drawback of this method is that we have to wait until the end of episode to acquire the actual value for $G_{t,T}$. TD solves this problem by using an estimate for this value. Namely, TD estimates $G_{t,T}$ as $R_{t+1} + \gamma V(S_{t+1})$, the sum of the observed immediate reward and the estimate of the next state's value. In other words, TD learning updates estimates for states based on other learned estimates, a process known as *bootstrapping*. Hence, we can write the **TD update rule** as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)],$$

where α is the learning rate and γ is the discount factor.

This TD method is called TD(0) or one-step TD and it is the most basic method of TD learning. It is a special case of TD(λ) and n-step TD methods which are not covered in this report, but can be found in [5].

The quantity in square brackets is called the TD error:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t).$$

If we look deeper in the TD(0) update rule, we can see that TD error finds the difference between current estimate of $V(S_t)$ and next estimate $R_{t+1} + \gamma V(S_{t+1})$, which is expected to be better than the current estimate. By this, it gives immediate feedback on the accuracy of the current value estimate $V(S_t)$. If $\delta_t > 0$ the agent underestimated the values of S_t, which means we need to adjust the value upwards, and vice versa. We multiply this value by a constant step-size α, which is between 0 and 1 to ensure convergence. In other words, we don't want the estimate of $V(S_t)$ to change rapidly because it may lead to big fluctuations. Then, we add this value to the current estimate of $V(S_t)$ to get the new estimate. By doing so, we are getting closer to our target, which is the estimated return for S_t.

Inspired by the style used in [5], we describe an algorithm for TD(0) below:

Tabular TD(0) for estimating v_π
Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}$, arbitrarily except that $V(\text{terminal}) = 0$

- Loop for each episode:
  - Initialize S
  - Loop for each step of episode:
    - Choose action A given by π for S
    - Take action A, observe R, S'
    - $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    - $S \leftarrow S'$

    until S is terminal

3.2 Efficiency and convergence of TD-learning

TD methods have certain advantages over Monte-Carlo and Dynamic Programming methods as mentioned at the beginning of this section. The first and most obvious advantage of TD methods is that they are naturally implemented in an online, fully incremental fashion. This means, unlike Monte Carlo methods, one does not have to wait until the end of an episode to update estimates. Instead, estimates can be used to improve other estimates as the episode continues. In many cases, this is a crucial feature, because many applications have very long episodes and waiting until the end of an episode slows down the learning process a lot. Some applications are continuing tasks and have no episodes at all.

Secondly, TD-learning is a model-free prediction method. That‘s why, it does not require a model of the environment, of its reward and next-state probability distributions. This flexibility gives TD-learning a strong advantage over Dynamic Programming methods, which require a model of the environment‘s dynamics. Because of this advantage, TD methods can be applied in environments where predicting or simulating the full environment is challenging or impossible.

It has been proven that TD(0) converges to v_π for any fixed policy π. With a constant but sufficiently small step-size parameter α, TD(0) converges in the mean. Furthermore, reducing the step-size parameter according to standard stochastic approximation guarantees almost sure convergence.

When it comes to the speed of convergence for TD and Monte Carlo

methods, there is no formal mathematical proof showing that one method converges faster than the other. However, in practical applications, TD methods generally reach convergence faster than constant-α Monte Carlo methods.

4 TD-control methods: SARSA and Q-Learning

So far, we talked about TD prediction problem, that is we tried to evaluate the values of states according to how much total reward we expect the agent to gain from them in the long run. However, this is only one part of the problem. As with improving our evaluations for returns we need to improve our policy, that is the way our agent acts or moves through states of the environment. This problem is addressed by TD Control methods. In this section, we explore these methods in detail.

4.1 Introduction to TD-Control

In reinforcement learning, control involves finding an optimal policy that maximizes the expected total reward for the agent. TD control problems can be divided into two categories: on-policy and off-policy TD control.

In **on-policy methods**, the agent tries to improve the policy it is currently following. In other words, target policy and the policy that the agent follows are the same. On-policy methods are generally simple and stable and useful for problems where it is crucial to balance exploration and exploitation within a single policy. The most common on-policy TD control method is SARSA.

In **off-policy methods**, the agent learns about a target policy while following a different policy to generate the data. This allows the agent to learn an optimal policy independently from the policy being used. Off-policy methods are generally expected to be more sample efficient and flexible in exploration. However, they may become unstable if one is not cautious. The most common off-policy TD control is Q-learning.

In Prediction problems, we considered transitions from state to state and tried to estimate values of states. In Control problems, we will be considering transitions from state-action pair to state-action pair and try to estimate the values of state-action pairs. This means we will be dealing with action-value functions rather than state-value functions.

4.2 SARSA update rule

As we mentioned earlier, SARSA is an on-policy TD control method. In SARSA, the agent updates its Q-values for (S_t, A_t) using the value of the next state-action pair S_{t+1}, A_{t+1} assuming the next action A_{t+1} is chosen according to **the current policy**.

SARSA stands for the sequence of terms that define its updates: the agent observes state S, takes action A, receives reward R, transitions to a new state S', from there takes a new action A'.

Definition 4.1 (SARSA Update rule)**.** SARSA update rule is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{4.1}$$

where α is the learning rate, γ is the discount factor, $Q(S_t, A_t)$ is the action value of state-action pair (S_t, A_t), A_{t+1} is the action chosen according to the agent's current policy π.

This means that, if the agent is in state S_t, and takes the action A_t and receives reward R_{t+1}, Q-value for this state-action pair is updated based on the action A_{t+1} that the agent will actually take following the current policy from the new state S_{t+1}. In words, SARSA tries to estimate the solution to **Bellman equation** for a given policy π. Therefore, Q-values estimated in SARSA converge to Q_π rather than Q.

4.3 Q-learning update rule

As we mentioned earlier, Q-learning is an off-policy TD-control method. In Q learning, the agent updates its Q-values using the maximum possible value of the next state S_{t+1} rather than the action chosen by the current policy. This means that Q-learning uses the value of the next state action pair assuming the next action is chosen **greedily**.

Definition 4.2 (Q-Learning)**.** Q-learning update rule is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right], \tag{4.2}$$

where $\gamma \max_a Q(S_{t+1}, a)$ is the maximum Q-value among all possible actions in the next state S_{t+1}.

In this equation, S_t is the current state and A_t is the action that the agent chooses at time t. α is the learning rate. It helps determine how much the new information affects the current Q-value. R_{t+1} is the reward

that we received after taking the action A_t at state S_t. γ is the discount factor, this helps with the weight of the future rewards. We will go through more regarding this below. $\max_a Q(S_{t+1}, a)$ is the highest Q-value for the next state S_{t+1}, in other words, the highest estimated future reward that is possible in the next state.

The key idea behind Q-learning is to approximate the solution to the **Bellman optimality** equation for action-value function. This means Q-learning directly tries to learn an optimal policy regardless of the current policy being followed.

This Q-learning update rule can help the agent to improve its understanding and knowledge of each action's reward from different states. So, if the agent keeps using this rule, the Q-values for each state-action pair become closer and closer to the optimal state-action pair values. Based on these values, the agent will be able to make optimal decisions.

Here is the **step by step algorithm for Q-learning**:

Initially, as we don't have knowledge about the environment, Q-values for all possible state-action pairs are given arbitrary values, except for the terminal state, which is given a value of 0. Then, the following is repeated for large number of episodes. For each episode, the starting point S is initialized. The following is done for every step of the episode until terminal state is reached. For each step, the agent selects an action from the current state based on some policy(e.g. ϵ-greedy) derived from action values. We will talk more about ϵ-greedy later on. Then the agent executes the action and moves to a new state and receives a reward. Using the Q-learning update rule, the corresponding action value is updated. Finally, the current state is updated since the agent has moved.

As the number of episodes increases, we will have more and more accurate values since more and more state-action pairs are visited by the agent. Therefore, the agent will be able to approximate the optimal strategy and to take the best possible actions for the corresponding state.

4.4 ϵ-greedy action selection

From what we know until now, Q-learning update rule is choosing the action that gives the maximum possible reward. However, as we know, Q-learning is a model-free reinforcement learning method. This means that we know nothing about the environment at the beginning episodes. So, what is best for the agent? Should the agent explore the environment? Probably yes,

because this will let the agent know about the possible actions and rewards. However, how much should be explored? The whole environment? If it is a huge state space then the time to explore the whole environment may take very long.

So, how does the agent decide which action to take? Here, we introduce the greedy action selection rule and the specific ϵ-greedy action selection rule that can be used in Q-learning.

Definition 4.3 (Greedy Action Selection Rule)**.** The simplest action selection rule is to act greedily. That is, to choose the action with the highest estimated value. Formally, at state S_t the agent chooses the action A_t, such that

$$A_t = argmax_a Q(S_t, a), \tag{4.3}$$

where $argmax_a(Q(S_t, a))$ is the action a for which $Q(S_t, a)$ is maximum.

This action selection rule uses the current knowledge to maximize the immediate reward. It will not consider the future effects of the actions, i.e. the long term reward. It only focuses on **exploitation** and does not give a chance for exploration. In other words, the agent only exploits its current knowledge and does not try to explore the environment for better rewards. Therefore, this rule cannot guarantee that the action chosen is the best one. To solve this issue, we introduce the ϵ-greedy action selection rule.

In the beginning episodes, we would probably want the agent to explore a lot to gain knowledge about the environment first. However, exploring takes a long time and leads to a low reward in many cases. That's why, we cannot only dedicate the agent to exploring. After some episodes, it will be better to use the current knowledge and gain high rewards. But before the whole environment is discovered, the agent will not be sure that the current knowledge produces the best reward. Therefore, we need to find such a balance that the agent is able to gain enough knowledge about the environment, but does not waste time exploring when it is not needed. This balance is hard to achieve and it can be different depending on the environment and how much of the environment we have already explored.

Therefore, we still need to exploit the current knowledge, but also give a chance to explore the environment. In order to achieve this, we can use the ϵ-greedy action selection rule.

Definition 4.4 (ϵ-greedy action selection)**.** ϵ-greedy action selection rule is to choose the action greedily with probability $1 - \epsilon$ and choose another

random action with probability ϵ. Formally, at state S_t, the agent chooses the action A_t, such that

$$A_t \leftarrow \begin{cases} \arg\max_a Q(S_t, a) & \text{with probability } 1-\epsilon, \\ \text{a random action} & \text{with probability } \epsilon. \end{cases}$$

Explanation:

- With probability $1-\epsilon$, the agent chooses the action for which $Q(S_t, a)$ is maximum. In other words, the agent exploits its current knowledge of the environment.
- With probability ϵ, the agent chooses a random action. In other words, it explores the environment.

By introducing the ϵ-greedy action selection, we can solve the issue above because we can control how much exploring is done for each episode by changing the ϵ value. Furthermore, we can change the value of ϵ as the number of episodes increases, so that the agent explores more in the beginning and then reduces the rate of exploration as the number of episodes increases. This can help the agent to achieve the highest reward throughout the process.

4.5 Comparison between SARSA and Q-Learning

We learned about SARSA and Q-Learning, the most popular on-policy and off-policy TD-control methods respectively. Now, the natural question is which one is better? The answer to this question is not simple. In fact, there is not really an answer to this question. As we mentioned in the introduction of the section, both methods have their strong sides and either one of them can be better than the other depending on the case. We will try to distinguish these strengths as well as weaknesses of these methods.

Cliff Walking Example (Example 6.6 in [5]) shows the difference between SARSA and Q-Learning very well. Imagine there is a grid world as shown in figure 4.1. The agent starts from state S and tries to reach the terminal state G. However, on the way, there is a "cliff" and if the agent falls off the cliff, they "die" and re spawn in state S again. We set up this problem as follows: The agent receives a reward of -100 if they move to The Cliff zone and immediately return to state S, a reward of -1 otherwise. There is no discount factor for future rewards.

Figure 4.2 shows the comparison of sum of the rewards per episode when this problem is solved using SARSA and Q-learning with ϵ-greedy action

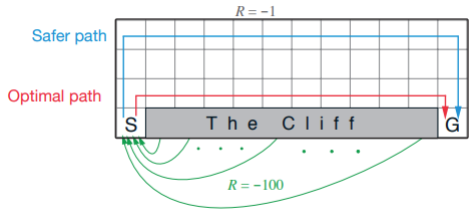

Figure 4.1: Grid world for the Cliff walking example[5].

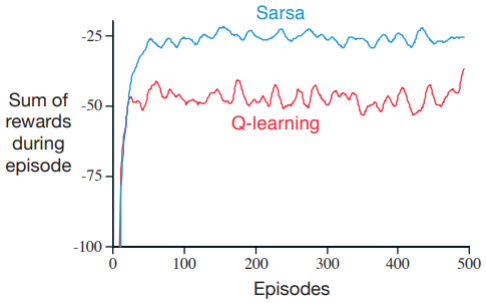

Figure 4.2: Comparison between the performance of SARSA and Q-Learning[5].

selection, $\epsilon = 0.1$. From the figures, we can see that Q-learning takes the optimal path as it learns the optimal policy, while SARSA takes a safer path, which is not optimal. However, the agent receives smaller sum of rewards in episodes, because the exploratory value $\epsilon = 0.1$ leads the agent to the cliff more often in Q-learning as the path chosen by the agent is closer to the cliff. SARSA considers this risk and takes a safer approach which is farther from the cliff. Furthermore, we can observe that the variance in terms of SARSA is smaller than Q-Learning. This difference can also be attributed to the safe approach chosen by SARSA, because the agent does not fall off the cliff and get a high negative reward as often as in Q-Learning approach. It should be noted that if ϵ were gradually reduced, then both methods would asymptotically converge to the optimal policy.

4.6 Double Q-learning

4.6.1 Bias issue in Q-Learning

In Q-learning, according to the update rule, maximum Q-value is used to update the current Q-value estimate for a state-action pair. This can lead to overestimating the Q-values. This bias is more affected when the environment's rewards are stochastic.

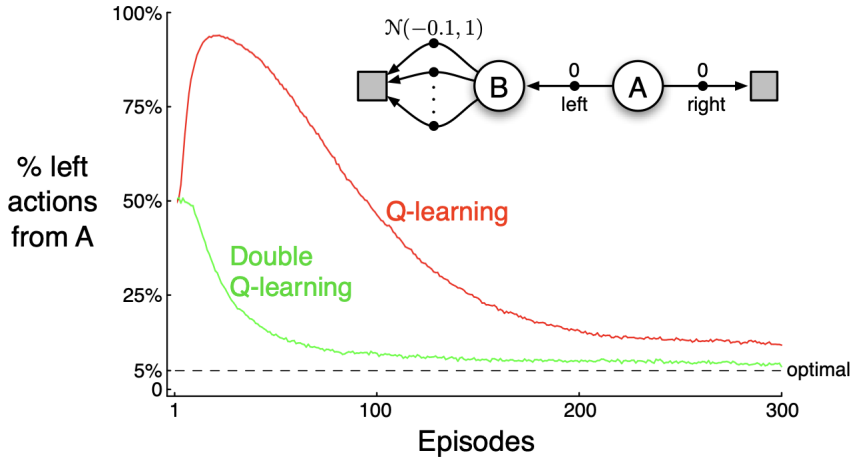

Figure 4.3: Comparison of Q-learning and Double Q-learning on Example 4.5. (Figure 6.5 in [5])

From the update rule, Q-learning assumes that the best estimate of the future rewards is the maximum reward of all the possible actions that you can take for the next step. However, if the rewards are stochastic, then some rewards may be inflated because they are sampled and the sample reward is higher than the average. If we keep choosing the maximum estimated Q-values and updating, then these errors will grow bigger and bigger, causing the Bias.

Example 4.5. (Example 6.7 in [5]) Consider an environment with two non-terminal states, A and B. The agent always starts in state A and can choose to go left or right. Going right transitions to a terminal state and gives a reward of 0. Going left transitions to state B and gives a reward of 0. From there, the agent has many actions to choose that all lead to a terminal state and give a reward taken from a normal distribution with mean -0.1 and variance 1.0.

From this, we know that the expected return of choosing the left is -0.1 and the right is 0. So taking the right is better than the left. However, Q-learning control leads the agent to choose the left more than the right because of the maximization bias making B more favorable than A.

When we use Q-learning with ϵ-greedy action selection rule, setting ϵ to 0.1, as seen in Figure 4.3, the agent tends to choose the left much more often than the right. Even at asymptote, the agent chooses the left 5% more than the optimal case. Therefore, now with this example, we can understand why Q-learning causes the maximization bias.

4.6.2 Double Q-learning

So, we know that one of the limitations of Q-learning is that it causes maximization bias when selecting optimal actions. So, how can we solve it?

In standard Q-learning, the agent updates the estimates of action values based on the maximum Q-value of the next state. However, if the estimations of action values are noisy, this leads to maximization bias because the agent both selects the action and evaluates its value using the same Q-function, potentially inflating the estimated values of actions. To solve this issue, we introduce the Double Q-learning method.

Double Q-learning uses 2 independent Q-values, Q_1 and Q_2. One of them is to find the maximizing action and the other is to estimate its value. These estimations are independent with each other but they use the same update rule. Hence, we use both estimates to make the final decision.

In a sense, double Q-learning divides the time steps into two parts. In each step, only one of the Q-values(Q_1, Q_2) is chosen to be updated each with a probability of 0.5. If Q_1 is chosen, the following update is made [2]:

$$\begin{aligned} Q_1(S_t, A_t) \leftarrow & Q_1(S_t, A_t) + \alpha \Big[R_{t+1} + \\ & + \gamma Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t) \Big]. \end{aligned}$$

If Q_2 is chosen, the following update is made:

$$\begin{aligned} Q_2(S_t, A_t) \leftarrow & Q_2(S_t, A_t) + \alpha \Big[R_{t+1} + \\ & + \gamma Q_1\big(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)\big) - Q_2(S_t, A_t) \Big]. \end{aligned}$$

It should be noted that both Q_1 and Q_2 can be used by the policy. For example, ϵ-greedy policy for double Q-learning can use the average or sum of these action value estimates to make decisions.

Figure 4.3 clearly shows that Double Q-Learning performs much better than Q-learning in the case of Example 4.5. From the start, the agent chooses the right much more often than the left because it is not affected by the maximization bias. As the number of episodes increases, the graph gets very close to the optimal value. This means that the agent is choosing left with a probability of slightly higher than 5 percent, which is imposed by exploration constant, $\epsilon = 0.1$.

5 Conclusion

We have talked about reinforcement learning which is a class of approximation methods for Markov Decision Processes based on optimization and Monte Carlo sampling. We first went through the MDP components including states, actions, rewards and transition functions. We also talked about the value function and optimal value function. These all helped to lay the foundation to understanding the Bellman Equation. Lastly, we mentioned the policy iteration and value iteration to end the section for MDPs.

In section 3, we talked about TD-learning which is a model-free learning method. For better understanding, we provided a comparison of TD methods with Monte Carlo and Dynamic Programming methods. We explained how the update rule of TD learning works and showed how to apply TD(0) as an algorithm. Furthermore, we talked about the efficiency and converge guarantee of TD-Prediction methods.

In section 4, we introduced two categories of TD-control methods: on-policy and off-policy methods. The main difference is that for on-policy methods, the agent focuses on improving the current policy while for off-policy methods, the agent focuses on finding the optimal policy. Then we focused on the most popular methods of these categories, SARSA and Q-Learning, by exploring their update rule.

Then we tried to solve the problem of balancing exploration and exploitation with ϵ-greedy action selection rule. This rule provides a way for us to determine how much the agent should explore. After that, we introduced the bias issue in Q-learning and introduced double Q-learning that could solve this issue.

Our project only covers the tip of the iceberg named reinforcement learning. A larger community within the field of reinforcement learning has conducted extensive research on topics such as Approximate Dynamic Programming, the CPI Algorithm, and the FQI Algorithm. However, due to time constraints and the level of difficulty, these topics could not be included in this project.

6 Group Team Work

We collaborated as a team throughout the project by holding both online and in-person meetings. We also arranged weekly meetings with our supervisor to receive feedback on the work we had completed. For quick communication and to discuss the progress of the report, we established a WhatsApp group. Throughout the project, we provided feedback on each other's work to ensure that the symbols and notations used were correct and consistent.

Work Allocation

After deciding on the structure and main sections of the project, we divided the workload among team members. Sections 1 and 5, the Introduction and Conclusion, were written and reviewed by all team members. Afiq wrote Section 2, which focuses on Understanding Discounted Reward MDPs. Section 3, covering TD-Learning, was written by Dovudkhon. Section 4, which discusses SARSA and Q-Learning, was shared between Dovudkhon and Arthur, with Arthur writing most of the section. Dovudkhon mainly worked on SARSA and the comparison between SARSA and Q-learning while Arthur worked on the other important parts.

References

[1] Sham M. Kakade W. Sun A. Agarwal, N. Jiang. *Reinforcement Learning: Theory and Algorithms*. 2022.

[2] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[3] S P Meyn. *Control Systems and Reinforcement Learning*. Cambridge University Press, 2022.

[4] Muddasar Naeem, Syed Rizvi, and Antonio Coronato. A gentle introduction to reinforcement learning and its application in different fields. *IEEE Access*, 2020.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.