Reinforcement Learning with Function Approximation methods

Dovudkhon Abdubokiev

Level: 3rd year, Credit Points: 20cp
Supervisor: Dr.Vladislav Tadic
Date of Submission: 19 February 2025

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
Date: 18 February 2025

Contents

Abstract

Reinforcement Learning is an important framework to make decisions in uncertain environments. Tabular RL methods, which represent value functions as tables, become practically infeasible in large or continuous state spaces because of computational and memory limitations. This report explores reinforcement learning with function approximation, an approach that addresses these issues by generalizing value functions using supervised learning. Firstly, we introduce the motivation and key concepts of function approximation. We then learn about on-policy prediction methods with function approximation, including semi-gradient TD(0) and linear methods, along with some theoretical results. Next, we learn about on-policy control methods with function approximation where we discuss semi-gradient SARSA and introduce an average-reward setting. Finally, we present a numerical example to compare tabular Monte Carlo and TD(0) performances.

1 Introduction to RL with Function Approximation

1.1 Overview of Tabular RL methods

In our Group Report(Appendix B), we covered Tabular Reinforcement Learning methods. In these methods, value functions and policies are represented as tables and each state has a dedicated entry in the table. These values are updated using different algorithms such as Dynamic Programming, Monte Carlo methods and Temporal Difference Learning. DP relies on full knowledge of the environment's transition dynamics, while MC estimates value functions using returns from completes episodes, and TD updates values using bootstrapping without waiting for an episode to finish. These methods are effective for small state spaces, but they become impractical as state space grows. Storing and updating every state value or action value in a table becomes will require substantial memory and computation time for large or continuous state spaces. That's why, we will need to use methods that generalize these values using supervised learning, which are called function approximation methods.

1.2 Motivation for Function Approximation

In most of the real-life problems, the state space is continuous or enormous and it is not physically possible to store and update all the state values in a table. In fact, many problems involve infinite state and action space. For example, in robotics, the position and velocity of a robot's arm can take infinitely many values. Similarly, the number of possible camera images is much larger than the number of atoms in the universe. In such cases, it is impractical to try to find optimal value function or policy even with the most sophisticated computers, so we have to settle with finding good approximations. To find these approximate solutions, we employ function approximation methods.

Function approximation is an example of supervised learning. Supervised learning methods take training data which consists of input-output examples and tries to generalize those examples when encountered with example that has not been seen. When we use function approximation with reinforcement learning, training data are examples of states and their calculated values by the agent with tabular methods. It is called function approximation because it takes example outcomes from a function and aims to approximate the entire function by generalization. Instead of representing the approximate value function as a table, function approximation uses parametrized functions,

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

with weight vector $\mathbf{w} \in \mathbb{R}^d$. The dimensionality of $\mathbf{w}$ is typically much smaller than the number of states. Hence, one weight can be assigned to many states, and changing one weight changes the estimated value of all the assigned states. Also, changing the value of one state can affect the value of many other states. While this generalization offers a very powerful advantage, it makes the learning more difficult to manage and balance.

2 On-policy Prediction with Approximation

In this section, we focus on the Prediction problem with function approximation. Here, the goal is to estimate the value function for a given policy. Firstly, we will introduce metrics to measure the accuracy of our approximations. Next, we will introduce gradient based methods followed by linear methods to approximate values. We will also present a detailed theoretical results for the convergence of semi-gradient TD(0) method under linear function approximation.

2.1 Mean Squared Value Error

Unlike tabular methods, state values in approximation methods do not equal the true value. That is why we need to establish some sort of identity to quantify how well our predictions are. As the number of weights is much lower than the number of states, changing the value of one state affects many other state values. As a result, if we try to improve the value of one state, we may make many other state values less accurate. That‘s why, we also need to decide which states are important to us, so that we can bring their values closer by sacrificing the precision of less important state values. For this purpose, we define $\mu(s) \geq 0, \sum_s \mu(s) = 1$, a state distribution that represents what fraction of time is spent on state s. In other words, $\mu(s)$ represents how much we care about the state s. Finally, our Prediction objective, Mean Squared Value Error ($\overline{VE}$) is given by:

$$\overline{VE}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2.$$

The expression inside the sum is the MSE for each state scaled by the importance of that state, $\mu(s)$.

Under on-policy training, which is the focus of this report, $\mu(s)$ is called the on-policy distribution and it is defined as the fraction of time spent in state s. In continuing tasks, it is simply the stationary distribution under policy π. Formally, it is defined as:

$$\mu(s) = lim_{t \to \infty} \mathbb{P}(S_t = s). \tag{2.1}$$

In episodic tasks, it depends on the distribution of initial states. Let $h(s)$ be the probability that an episode starts in state s, let $\eta(s)$ be the average number of time steps spent in state s in a single episode. Time is spent in state s either when an episode begins in s, or when the agent reaches s by transitioning from a previous state $\bar{s}$ during an episode. Then, we have

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(s) \sum_{a} \pi(a|\bar{s}) p(s|\bar{s}, a), \text{ for all } s \in \mathcal{S}.$$

On-policy distribution is just the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s)}, \text{ for all s} \in \mathcal{S}.$$

Although, the formulation of $\mu(s)$ is different for continuing and episodic tasks, it signifies a similar meaning and behaves similarly in both cases.

Intuitively, we would think that our objective is to try to make $\overline{VE}$ as small as possible, perhaps find a global optimum, a weight vector $\mathbf{w}^*$ for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all possible $\mathbf{w}$. However, this is not always possible or even desirable. As we have seen with other supervised learning methods, the predictor that minimizes $\overline{VE}$ is not necessarily the best predictor for our value function. Additionally, in complex function approximators such as artificial neural networks, decision trees, it is not always possible to find a global minimum. In many cases, we will be satisfied with local minimum, a weight vector $\mathbf{w}^*$ for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$.

2.2 Gradient-Based Methods for Prediction

Let us remind ourselves that our objective is to minimize the State Value error, $\overline{VE}(\mathbf{w})$. We now introduce gradient-descent methods to solve this problem. As a starter, we introduce a notation $\mathbf{w}_t$, weight vector at time step t. We assume that states in example or training data have the same distribution, μ. We try to reduce $\overline{VE}$ on the observed examples using *Stochastic gradient-descent*(SGD). This method involves, for each example adjusting the weight vector by a small amount in the direction of the negative gradient:

$$\begin{aligned} \mathbf{w_{t+1}} &= \mathbf{w_t} - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w_t} + \alpha [v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla v(s_t \hat{,} \mathbf{w}_t) \end{aligned} \tag{2.2}$$

where $\alpha > 0$ is a step-size parameter, and $\nabla f(\mathbf{w})$ is the gradient of f with respect to $\mathbf{w}$:

$$\nabla f(\mathbf{w}) = \left(\frac{\delta f(\mathbf{w})}{\delta w_1}, \frac{\delta f(\mathbf{w})}{\delta w_2}, \ldots, \frac{\delta f(\mathbf{w})}{\delta w_d} \right)^{\mathsf{T}}.$$

However, in practice, the true value function $v_\pi(s)$ is unknown and the error cannot be computed exactly. To make the SGD update practical, we need to find some approximation U_t, instead of the true value $v_\pi(S_t)$. Then, we will be able to use that approximation U_t in equation 2.2 in place of $v_\pi(S_t)$. Given that $\mathbb{E}[U_t|S_t = s] = v_\pi(S_t)$, for each t, that is, if U_t is unbiased, $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α, which are given below:

Let α_n be the step-size parameter after the nth step. Then, the usual convergence conditions are:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty. \tag{2.3}$$

One popular example of U_t is the Monte Carlo target G_t. G_t is an unbiased estimate of $v_\pi(S_t)$ because the true value of a state is the expected value of the return following it. This was discussed in Section 2.3 of our Group Report(Appendix B). This guarantees the convergence of $\mathbf{w}_t$ to a local optimum. Pseudocode for the complete algorithm(Page 202 in [5]) and its explanation is given below:

```
Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$
Set step size parameter $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)

Loop for each episode:
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \dots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,(G_t - \hat{v}(S_t, \mathbf{w}))\,\nabla\hat{v}(S_t, \mathbf{w})$
```

Explanation: The algorithm receives the policy π and the value function $\hat{v}$ as inputs. We need to set the step-size α and initialize a vector to store our estimation of $\mathbf{w}$. For each episode, we calculate the return G_t once the episode finishes, and update $\mathbf{w}$ using this return and the update rule in 2.2.

Another common version of U_t is the bootstrapping estimate of $v_\pi(S_t)$. We also covered examples of this estimates in Sections 2.6 and 3.1 of our Group Report(Appendix B). However, bootstrapping target such as TD(0) target, $G_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$ or the DP target, $\sum_{a,s',r} \pi(a|S_t)p(s', r|S_t, a)[r + \gamma\hat{v}(s', \mathbf{w}_t)]$ are biased and do not produce a true gradient-descent method, because they depend on the current value of the weight vector $\mathbf{w}_t$. They care about the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target. They only include a part of the gradient and therefore called the *semi-gradient methods.*

Although semi-gradient methods do not provide robust convergence as gradient methods, they still produce reliable results. Moreover, they offer a big computational advantage. As estimates can be improved using other estimates, the learning process is much faster in semi-gradient methods. Furthermore, they enable online learning without waiting for an episode to finish. Below, we provide a complete algorithm for semi-gradient TD(0)(page 203 in [5]) and its explanation:

```
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : S^+ \times \mathbb{R}^d \to \mathbb{R}$
such that $\hat{v}(terminal, \cdot) = 0$
Set step size parameter $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action $A$, observe $R, S'$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})\right) \nabla \hat{v}(S, \mathbf{w})$
        $S \leftarrow S'$
    until $S$ is terminal
```

Explanation: The algorithm receives the policy π and the value function $\hat{v}$ as inputs. We need to set the step-size α and initialize a vector to store our estimation of $\mathbf{w}$. For each step of each episode, the agent chooses an action A following the policy, takes the action and receives reward R and moves to a new state S'. The estimate of $\mathbf{w}$ is then updated using the TD(0) target in the SGD update rule (Equation 2.2). S is updated to S' that was observed. In the final step, we check if S is terminal.

We can explore the following example, to better understand the difference between the two methods introducesd: Stochastic gradient-descent and semi-gradient descent. For this, we first need to introduce *state aggregation*, which is a simple form of function approximation. It works by grouping several states together and assigning one weight to them. The values of states in one group are all the same.

Example 2.1 (Example 9.1 in [5])**.** Let's consider a random walk with 1000 states, which are numbered 1 to 1000 from left to right. The agent always starts from the 500th state. From the current state, the agent can move to any of the 100 states to the left or any of the 100 states to the right,

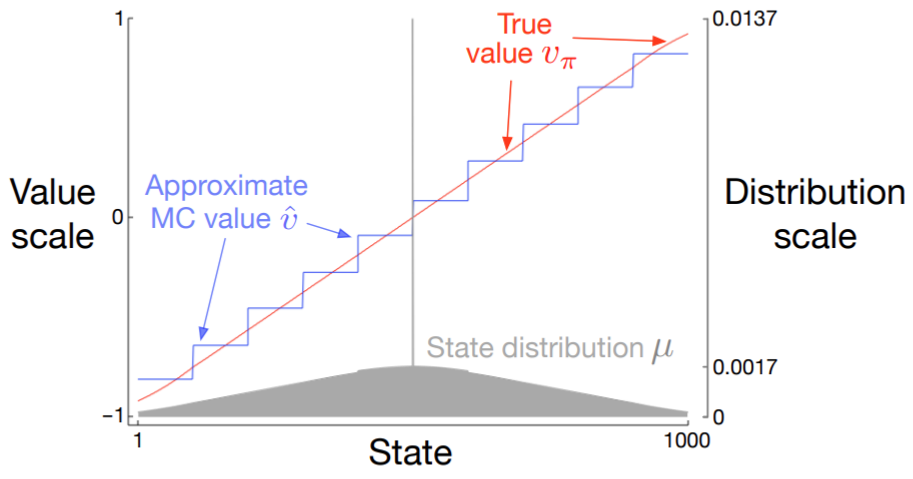

Figure 2.1: Function approximation by state aggregation on the 1000-state Random Walk task using the gradient Monte Carlo algorithm. (Figure 9.1 in [5])

all with equal probability. If the agent is near the edge and has fewer than 100 available states to transition to, then the probability of all the missing states will be given to terminating on that side of the random walk. For example, if the agent is in state 50, then the probability of terminating on the left will be $\frac{100-50}{200} = \frac{50}{200}$. The same applies if the agent is in the edge on the right side. Terminating on the left returns -1 reward, while terminating on the right returns +1 reward. All other transitions return a zero reward.

In Figure 2.1, we can see the comparison between fitted values using gradient Monte-Carlo algorithm with state aggregation and the true state function, v_π, of the 1000-state Random Walk introduced above. We can see that our fitted values are approximating the true values quite closely. The given approximate values are achieved after 100,000 episodes with a step size, $\alpha = 2 \times 10^{-5}$. The states were divided into 10 groups of 100 states each. The step like shape is due to the states in one group all receiving the same state values.

Another important detail in this graph is the state distribution, μ, whose scale is on the right side of the graph. We can see that there is a tall spike at state 500. This is because every episode starts from this state and on average 1.37% of the time steps is spent in this state. Second most visited states are the ones the agent can reach in one transition from the starting state, and about 0.17% os the time is spent in each of these states. The most insteresting effect can be observed in the left-most or right-most groups. If we look at the group of 1-100 states, we can see that the group's value is higher than the unweighted average of the true values of the states in that

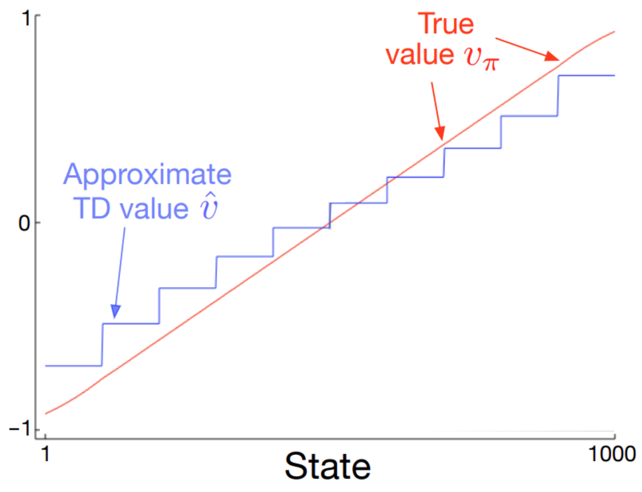

Figure 2.2: Function approximation by state aggregation on the 1000-state Random Walk task using the semi-gradient TD(0) algorithm. (Figure 9.2 in [5])

group. This is because more time is spent in states which are numbered higher and the algorithm is sacrificing the accuracy of the less visited states to get a better accuracy for more visited states.

In Figure 2.2, we can see the comparison between between fitted values using semi-gradient TD(0) algorithm with the same settings as above and the true state function, v_π. As discussed, we see that the semi-gradient TD(0) is indeed biased as the estimated values are farther from the true values than the gradient Monte-Carlo approximation shown in Figure 2.1. However, as semi-gradient methods offer strong computational advantages, they are still widely used.

2.3 Linear Methods

In this subsection, we will explore one of the most important instances of function approximation, where the approximate function, $\hat{v}(\cdot, \mathbf{w})$, is a linear function of the weight vector, $\mathbf{w}$. In this method, for every state, there is a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), ..., x_d(s))^\intercal$, where d is the dimensionality of the weight vector, $\mathbf{w}$. Approximate state-value function is given by:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\intercal \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s). \tag{2.4}$$

The gradient of the approximate value function with respect to $\mathbf{w}$ in this case is:

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

Hence, the general SGD update (2.2) in the case of linear function approximation becomes:

$$\begin{aligned}\mathbf{w_{t+1}} &= \mathbf{w_t} + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla \hat{v(s_t, \mathbf{w}_t)} \\ &= \mathbf{w_t} + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t)\end{aligned}$$

Because of its simplicity, the linear SGD case is favored for mathematical analysis in many cases. Almost all useful convergence results are for linear function approximation methods. In the linear case, there is only one optimum. Therefore, if a method converges to or near a local optimum, then it automatically converges to the global optimum.For example, the gradient Monte-Carlo algorithm discussed above converges to the global optimum of the $\overline{VE}$ in the case of linear function approximation given that α is reduced over time according to the usual conditions given in (2.3).

The semi-gradient TD(0) discussed above also converges to a point close to local optimum under linear function approximation. Below we give a detailed derivation of the weight vector $\mathbf{w}_{TD}$ it converges to:

For continuing tasks, the update at each time t is given by:

$$\begin{aligned}\mathbf{w_{t+1}} &= \mathbf{w_t} + \alpha(R_{t+1} + \gamma \mathbf{w}_t^\intercal \mathbf{x}_{t+1} - \mathbf{w}_t^\intercal \mathbf{x}_t)\mathbf{x}_t & (2.5) \\ &= \mathbf{w_t} + \alpha(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\intercal \mathbf{w}_t), & (2.6)\end{aligned}$$

where we use the notation $\mathbf{x}_t = \mathbf{x}(S_t)$.

For any weight vector $\mathbf{w}_t$, once the system reaches the steady state, we can write the expected next weight vector in the following way:

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \quad (2.7)$$

where

$$\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \text{ and } \mathbf{A} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\intercal] \in \mathbb{R}^{d \times d}$$

Looking at 2.7, we can see that for the system to converge to some weight vector $\mathbf{w}_{TD}$, the following should be satisfied:

$$\begin{aligned}\mathbf{b} - \mathbf{A}\mathbf{w}_t &= 0 \\ \mathbf{b} &= \mathbf{A}\mathbf{w}_{TD} \\ \mathbf{w}_{TD} &= \mathbf{A}^{-1}\mathbf{b}. & (2.8)\end{aligned}$$

$\mathbf{w}_{TD}$ is called the TD fixed point and the semi-gradient TD(0) converges to this point. Below, we provide proof for the convergence:

The equation 2.7 can be rewritten as:

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \tag{2.9}$$

We first note that $\mathbf{b}$ is not multiplied by $\mathbf{w}_t$, so we only need to consider the matrix $\mathbf{A}$. Let's first cosider the special case in which the matrix $\mathbf{A}$ is diagonal. Then, $\mathbf{I} - \alpha\mathbf{A}$ is also diagonal. Furthermore, it can be noted that if any of the diagonal elements of $\mathbf{I} - \alpha\mathbf{A}$ is greater than one, it will amplify the corresponding component of $\mathbf{w}_t$. This will lead to divergence if continued. On the other hand, if all the diagonal elements of $\mathbf{I} - \alpha\mathbf{A}$ is between 0 and 1, it shrinks $\mathbf{w}_t$ and this guarantees stability. Therefore, we need all the diagonal elements of $\mathbf{I} - \alpha\mathbf{A}$ to be between 0 and 1.

If any of the diagonal elements of the matrix $\mathbf{A}$ is negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one regardless of the value of α. That's why, we need all the diagonal elements of the matrix $\mathbf{A}$ to be positive. In this way, we can choose some $\alpha < 1$, so that all the diagonal elements of $\mathbf{I} - \alpha\mathbf{A}$ is between 0 and 1.

In general, to achieve stability in $\mathbf{w}_t$ the matrix $\mathbf{A}$ must be positive definite. In other words, $y^\intercal \mathbf{A} y > 0$ for any real vector $y \neq 0$. The existence of the inverse $\mathbf{A}^{-1}$ is also assured, given the matrix $\mathbf{A}$ is positive definite.

In the case of linear TD(0), in the continuing case with $\gamma < 1$, the $\mathbf{A}$ matrix can be written as:

$$\begin{aligned}
\mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a \mid s) \sum_{r,s'} p(r, s' \mid s, a)\mathbf{x}(s)(\mathbf{x}(s) - \gamma\mathbf{x}(s'))^\top \\
&= \sum_s \mu(s) \sum_{s'} p(s' \mid s)\mathbf{x}(s)(\mathbf{x}(s) - \gamma\mathbf{x}(s'))^\top \\
&= \sum_s \mu(s)\mathbf{x}(s) \left(\mathbf{x}(s) - \gamma \sum_{s'} p(s' \mid s)\mathbf{x}(s')\right)^\top \\
&= \mathbf{X}^\top \mathbf{D}(\mathbf{I} - \gamma\mathbf{P})\mathbf{X},
\end{aligned}$$

where $\mu(s)$ is the stationary distribution under policy π (defied in 2.1), $p(s'|s)$ is the probability of transitioning from state s to state s' under π. In the final expression, $\mathbf{P}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix composed of these transition probabilities, $\mathbf{D}$ is the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix, whose diagonal elements are the values of stationary distribution, $\mu(s)$, and $\mathbf{X}$ is the $|\mathcal{S}| \times d$ matrix whose rows are $\mathbf{x}(s)$. The final matrix form of the equation clearly shows that we need to consider the key matrix in the middle, $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$, to determine the positive definiteness of $\mathbf{A}$.

For a key matrix of this form, positive definiteness is assured if all of its columns sum to a nonnegative number. This was established by Sutton(1988, p. 27) and we use this theorem without proof here. Therefore, we only need to show that the sum of the elements of each column of the matrix, $\mathbf{D}(\mathbf{I} - \gamma \mathbf{P})$, is nonnegative.

We can write the row vector containing the column sums of any vector, $\mathbf{M}$, as $\mathbf{1}^\top \mathbf{M}$, where $\mathbf{1}$ is the column vector all of whose elements are 1. Let μ be the $|\mathcal{S}|$-vector of the $\mu(s)$, where $\mu = \mathbf{P}^\intercal \mu$ because μ is the stationary distribution. Then, the column sums of our key matrix can be expressed as:

$$
\begin{aligned}
\mathbf{1}^\top \mathbf{D}(\mathbf{I} - \gamma \mathbf{P}) &= \boldsymbol{\mu}^\top (\mathbf{I} - \gamma \mathbf{P}) \\
&= \boldsymbol{\mu}^\top - \gamma \boldsymbol{\mu}^\top \mathbf{P} \\
&= \boldsymbol{\mu}^\top - \gamma \boldsymbol{\mu}^\top \quad \text{(because } \boldsymbol{\mu} \text{ is the stationary distribution)} \\
&= (1 - \gamma) \boldsymbol{\mu}^\top .
\end{aligned}
$$

We know $\gamma < 0$ and all components of μ is positive. Therefore, it is easy to see that all the components of the final vector are positive. Therefore, we can conclude that the key matrix, $\mathbf{D}(\mathbf{I} - \gamma \mathbf{P})$, and the matrix $\mathbf{A}$ are positive-definite and hence the semi-definite TD(0) is stable. $\square$

In the continuing case, it has also been shown that the $\overline{VE}$ at the TD fixed point, $\mathbf{w}_{TD}$, is bounded by the lowest possible error scaled by $\frac{1}{1-\gamma}$:

$$
\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{VE}(\mathbf{w}) \tag{2.10}
$$

Here, the lowest possible error, $\min_{\mathbf{w}} \overline{VE}(\mathbf{w})$, is achieved in the limit by Monte Carlo method. This looks like a promising bound, but γ is usually close to one. Therefore, this bound can be really large, so the asymptotic performance of TD method is still much less precise than Monte Carlo method.

3 On-policy Control with Approximation

In this section, we move on to learning the On-policy Control with Approximation. Control Problem is focused on finding the optimal policy by finding optimal action values. In this section, we will now focus on the parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector with finite dimension d. We will consider the extension of the semi-gradient TD(0) to action values, which is the semi-gradient SARSA. We will consider the episodic and continuing cases separately. While the episodic case is straightforward, the continuing case requires giving up the discounting of rewards and instead using a new setting, average-reward setting and new differential value functions.

3.1 Semi-gradient Control: Episodic case

As we have mentioned, the extension of the prediction problem to the control problem in episodic cases is straightforward. If we considered random training examples of the form $S_t \to U_t$ in the previous section, we now consider examples of the form $S_t, A_t \to U_t$. This corresponds to using action-value functions instead of state-value functions. We can use the usual approximate values for $q_\pi(S_t, A_t)$ that we studied in Section 4 of our Group Report(Appendix B) as the target U_t. Some examples are the Monte-Carlo target or the SARSA target.

The update rule of the weight vector is also similar to the prediction case(2.2). We only need to switch the state-value functions with action-value functions. The general gradient-descent update is as follows:

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{3.1}$$

A special example of this general update rule is the *episodic semi-gradient one-step SARSA*:

$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t). \tag{3.2}$$

Like the semi-gradient prediction, this method also converges similarly under a constant policy. Moreover, it guarantees the same kind of error bound as we stated in (2.10).

Below is psudocode for the complete episodic semic gradient SARSA(page 244 in [5]) and its explanation:

```
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Set step size parameter $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R, S'$
    If $S'$ is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R - \hat{q}(S, A, \mathbf{w})\right] \nabla \hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\right] \nabla \hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$
```

Explanation: We first select suitable α for step size and ϵ for the exploration parameter. After initializing the weight vector arbitrarily, the following steps are repeated for each episode. Firstly, the agent selects the initial state and action by following the ϵ-greedy policy to start the episode. Then, for each step of the episode, it does the following. The agent takes the action A that was selected in the previous step, receives reward R and moves to a new state S'. If the new state is terminal, we use only the current reward as the target to update $\mathbf{w}$ and move on to a new episode. Otherwise, it continues the episode and selects a new action from the new state according to the policy. Then, the weight vector is updated using the episodic semi-gradient SARSA update rule. In the last line, the state and action is updated as well.

3.2 Semi-gradient Control: Continuing case

As we mentioned earlier, the discounted setting creates challenges with function approximation. That's why, we now introduce a new problem setting for the control problem in continuing tasks: the *average-reward* setting. In this setting, there is no discounting, which means that the rewards received later are just as important as the immediate rewards. In fact, as we see later, average-reward setting and the discounted setting will produce the same policy ranking as γ approaches 1.

We firstly introduce a new term called the *average reward*, which is used to assess how good a certain policy π is. Hence, the average reward, $r(\pi)$, intuitively is a function of π and it is expressed as follows:

$$\begin{align} r(\pi) &= lim_{h\to\infty}\frac{1}{h}\sum_{t=1}^{h}\mathbb{E}[R_t|S_0, A_{0:t-1}\sim\pi] \tag{3.3}\\ &= lim_{t\to\infty}\mathbb{E}[R_t|S_0, A_{0:t-1}\sim\pi] \tag{3.4}\\ &= \sum_s \mu(s)\sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)r, \tag{3.5}\end{align}$$

where S_0 is the initial state and $A_{0:t-1} : A_0, A_1, \ldots, A_{t-1}$ are the actions taken under the policy π. $\mu(s)$ is the steady-state distribution defined in the previous section. Now we have a quantity that we can use to rank different policies. We can say one policy is better than the other if the average reward under that policy is higher. If several policies achieve the maximal value of $r(\pi)$, then we consider all of them as optimal policies.

Now we introduce the term called the *differential return*, which is defined as the sum of differences between rewards and the average-reward:

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \ldots. \tag{3.6}$$

Using the differential returns, we introduce the differential state-value and action-value functions. These value functions are defined in the same way as we defined the value functions in Section 2.3 of our Group Report(Appendix B). The only difference is that we use the differential return instead of the discounted return. These differential value functions also support the Bellman equations which are defined in a very similar way to the Bellman equations defined in Sections 2.4, 2.5 of our Group Report as well. We should just remove the discounting factor and replace the rewards with the difference between the rewards and the average reward:

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{r,s'} p(s',r|s,a)\left[r - r(\pi) + v_\pi(s')\right],$$

$$q_\pi(s) = \sum_{r,s'} p(s',r|s,a)\left[r - r(\pi) + \sum_{a'}\pi(a'|s')q_\pi(s',a')\right].$$

Similarly, we can define the Bellman Optimality Equations with just small adjustments:

$$v_*(s) = \max_a \sum_{r,s'} p(s',r|s,a)\left[r - \max_\pi r(\pi) + v_*(s')\right],$$

$$q_*(s) = \sum_{r,s'} p(s',r|s,a)\left[r - \max_\pi r(\pi) + \max_{a'} q_*(s',a')\right].$$

Furthermore, we can introduce the differential form of the TD error, covered in Section 3.1 of our Group Report(Appendix B), by making similar changes:

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \tag{3.7}$$

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{3.8}$$

where $\bar{R}_t$ is an estimate of the average reward $r(\pi)$ at time t.

These new definitions can be used in many algorithms and theorems in the average-reward setting in a similar way to the discounted-reward setting. One example of this is the *differential semi-gradient SARSA*, which is the average-reward version of the semi-gradient SARSA. Pseudocode and its explanation is presented below:

```
Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state $S$, and action $A$
Loop for each step:
    Take action $A$, observe $R, S'$
    Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
    $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$
```

Explanation: We choose appropriate step sizes $\alpha, \beta > 0$ and the exploration parameter, ϵ. α is used in the updating of the weight vector and β is used in the updating of the average reward estimate. We then initialize $\mathbf{w}$ and $\bar{R}$ arbitrarily. As this is the continuing case, there is no loop over each episode. After initializing the first state S and action A, the agent then does the following for each step. It takes the action A chosen in previous line, receives reward R and moves to a new state, S'. Then, it chooses a new action A' based on the policy and the q-function. Then, δ is calculated using the differential definition of the TD-error. The estimate for the average reward is improved by adding δ scaled by the learning rate β. In the next line, the estimate for the weights is improved using the general update rule, but with the differential TD-error. In the last two lines, the state and action are updated accordingly.

3.3 Why we don't need discounting

In the tabular setting, the discounted reinforcement learning has been very useful because each state can be separately identified and averaged. However, in the case of function approximation, states are not explicitly represented, but encoded as feature vectors. As an extreme case, even all of the feature vectors may be the same, leaving us with nothing but a sequence of rewards and actions. In this case, natural way of assessing performance would be based on the long-term average reward. Even if we introduce discounting, we would still have to average the returns over a large enough time interval. Otherwise, the values would fluctuate a lot and we would not obtain a meaningful performance measure.

Surprisingly, the average of the discounted returns is proportional to the average reward. This can be seen using a symmetry argument. We first note that all time steps are exactly the same. Then we notice that with discounting every reward is a component of some return. In different returns, this reward is discounted by every degree of γ exactly once. For example, the tth reward will appear undiscounted in the $t-1$st return, discounted by γ in the return at $t-2$, discounted by γ^2 in the return at $t-3$, and so on. Therefore, the tth reward is weighted by $1+\gamma+\gamma^2+\gamma^3+\cdots=\frac{1}{1-\gamma}$. Since, all states are the same, the average of the discounted returns ends up being exactly $\frac{r(\pi)}{1-\gamma}$. When γ is close to one, this is basically the average-reward, $r(\pi)$.

The above result means that even if we introduced discounting, the ranking of policies would only depend on the average reward $r(\pi)$. No matter the value of γ, we would still obtain the same policy ranking. That's why, it can be concluded that discounting has no role in the definition of the control problem with function approximation.

4 Numerical Example

In this section, we compare the prediction performance of TD(0) and constant-α Monte Carlo methods in terms of the following numerical example. Let's consider a random walk with 9 states. States are numbered 0 to 8 from left to right and every episode starts from the middle state, 4. At every state, there is an equal probability of going left or right and the states 0 and 8 are terminal. Terminating at state 8 yields a reward of 1, all other transitions yield a reward of 0. The discounting factor, γ, is one, so no discounting. This example is similar to Example 6.2 from [5], but we now have 9 states instead of 7.

Firstly, let's find the true value of each state to be able to compare the predictions of methods later. As there is no discounting, the true value of each state is the same as the probability of terminating at the state 8

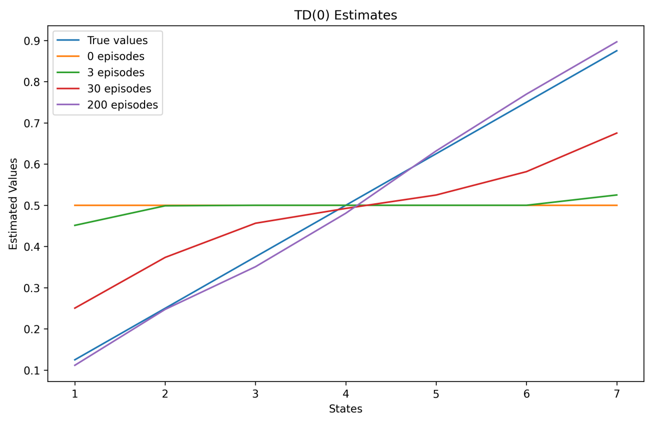

Figure 4.1: True state values vs the TD(0) Estimates obtained using our Python program

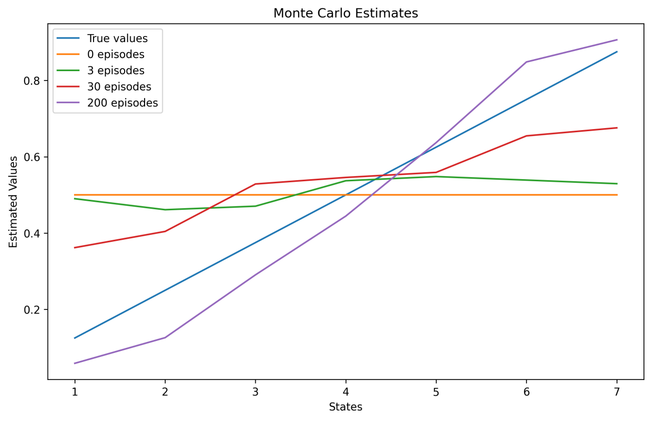

Figure 4.2: True state values vs the Monte Carlo Estimates obtained using our Python program

if starting from that state. The probability of terminating on the right if starting from the middle state 4 is 4/8, $v_\pi(4) = 0.5$. The true values of the states from 1 to 7 are: $1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8$.

Using python, we implemented the Pseudocode that we discussed in Section 3.1 of our Group Report (Appendix B) for the Tabular TD(0) algorithm. We ran our algorithm for 200 episodes and plotted the state value estimates after 3, 30 and 200 episodes vs the true state values. By doing this, we were able to obtain Figure 4.1 and see how the state value estimates change as the number of episodes increases.

We did the same with the Monte Carlo method and we obtained Figure 4.2.

From the figure, we can already see that TD(0) estimates are converging

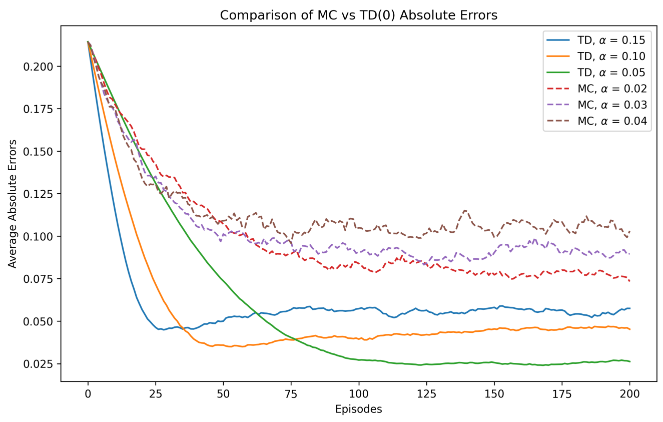

Figure 4.3: Comparison of constant-α MC and TD(0) using mean absolute errors

faster than MC estimates. After 200 episodes, TD(0) estimates are very close to the true values, but MC estimates are still rather far from the true values. This can be attributed to the *bootstrapping* feature of TD(0). As it uses estimates to update other estimates, it does not have to wait for an episode to finish to make updates. It can update the values in an online fashion during the episode. On the other hand, MC method has to wait for an episode to finish, so that it can make the updates using the real value of the return at the end of the episode. That's why the TD(0) estimates converge faster than the MC estimates

Next, we tried to learn this difference in performance even deeper and more accurately. For this, we ran our algorithm 200 times for different values of step size, α, and calculated the mean absolute error for every state during each episode and further averaged this value by the number of states. The result can be seen in Figure 4.3.

From the figure, we can see once more that TD(0) is faster at learning than MC. The TD(0) error values are plummeting already in the beginning episodes, while the MC errors are decreasing rather slowly. Furthermore, even after 200 steps, for all values of α, TD(0) error values are lower than those of MC. Therefore, for this task, we can conclude that TD(0) is performing better than MC at prediction accuracy and the speed of convergence.

5 Conclusion

In this report, we tried to address the limitations of tabular methods in large or continuous state spaces by exploring RL with function approximation methods. We began by discussing the limitations of tabular RL methods in real-world problems and how function approximation addresses these issues.

In the prediction problem, we introduced Mean Squared Value Error to assess the accuracy of our approximations. We learned different prediction methods with function approximation such as semi-gradient TD(0). We also went through examples and convergence properties of the gradient-based and semi-gradient methods in linear case.

For the control problem,we extended the prediction methods to action-value functions, focusing on on-policy control methods like semi-gradient SARSA. We also introduced a new setting: average-reward setting, in addition to episodic and discounting settings, for formulating the goal in Markov Decision problems.

Through a numerical example, we demonstrated the comparative performance of Temporal Difference Learning and Monte Carlo methods. We once again confirmed that bootstrapping gives a strong computational advantage to Temporal Difference methods.

Overall, function approximation plays a big role in scaling and generalizing reinforcement learning algorithm, making them applicable to real-world problems with large and complex state and action spaces.

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.

A Python code for the Random Walk Example

```python
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

seed = 21641 #for reproducibility
np.random.seed(seed)

#Initialixing true values and initial values as lists
true_V = [0/8, 1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8, 8/8]
initial_V = [0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0]

def td_method(V, alpha = 0.05):
    state = 4 #episode always starts at the middle state
    while True:
        action = np.random.choice([-1, 1]) #action A - going right or left
        new_state = state + action #observe new state S'
        reward = 1.0 if new_state == 8 else 0.0 #observe reward R

        #make the TD(0) update
        V[state] = V[state] + alpha * (reward + V[new_state] -V[state])
        state = new_state #update the state value: S<-S'
        #check if S is terminal
        if state == 0 or state == 8:
            break

def mc_method(V, alpha = 0.02):
    state = 4 #epsisode always start at the middle state
    state_sequnce = [state]
    returns = 0.0
    while True:
        action = np.random.choice([-1, 1]) #action A - going right or left
        state += action #observe new state S'
        state_sequnce.append(state)
        #check if S is terminal and calculate returns
        if state == 0:
            returns = 0.0
            break
        elif state == 8:
            returns = 1.0
            break
    #make the Monte Carlo update
    for state in state_sequnce[:-1]:
        V[state] = V[state] + alpha * (returns - V[state])

def calculate_state_values(method = 1):
    #plot the true values
    plt.plot(("1", "2", "3", "4", "5", "6", "7"), true_V[1:8], label='True values')
    #plot the changing estimates as the number of episodes increses
    n_episodes = [0, 3, 30, 200]
    current_V = initial_V.copy()
    for i in range(201):
        if i in n_episodes:
            plt.plot(("1", "2", "3", "4", "5", "6", "7"),\
                    current_V[1:8], label=str(i) + ' episodes')
        td_method(current_V) if method == 'TD' else mc_method(current_V)
    plt.xlabel('States')
    plt.ylabel('Estimated Values')
    plt.legend()

def calculate_mean_abs_error():
    td_alphas = [0.15, 0.1, 0.05]
    mc_alphas = [0.02, 0.03, 0.04]
    n_episodes = 201
    runs = 200
    #Calculate td(0) errors for different values of alphas
    for alpha in td_alphas:
        #array to store state-averaged errors for each episode averaged over runs
        total_errors = np.zeros(n_episodes)
        for _ in range(runs):
            #array to store errors for each episode averaged over states
            errors = []
            current_V = np.copy(initial_V)
            for _ in range(n_episodes):
                #append the error averaged over 7 states
                errors.append(np.sum(np.abs(true_V[1:8] - current_V[1:8]))/ 7.0)
                td_method(current_V, alpha=alpha)
            #sum the errors for every episode when a run is complete
            total_errors += np.asarray(errors)
        total_errors /= runs #average errors over runs
        plt.plot(total_errors, linestyle='solid',\
            label= 'TD, $\\alpha$ = %.02f' % (alpha))
    for alpha in mc_alphas:
        #array to store state-averaged errors for each episode averaged over runs
        total_errors = np.zeros(n_episodes)
        for _ in range(runs):
            #array to store errors for each episode averaged over states
            errors = []
            current_V = np.copy(initial_V)
            for _ in range(n_episodes):
                #append the error averaged over 7 states
                errors.append(np.sum(np.abs(true_V[1:8] - current_V[1:8])) / 7.0)
                mc_method(current_V, alpha=alpha)
            #sum the errors for every episode when a run is complete
            total_errors += np.asarray(errors)
        total_errors /= runs #average errors over runs
        plt.plot(total_errors, linestyle='dashed',\
                label= 'MC, $\\alpha$ = %.02f' % (alpha))
    plt.xlabel('Episodes')
    plt.ylabel('Average Absolute Errors')
    plt.legend()

def plot_figures():
    # First plot: MC Estimates
    plt.figure(figsize=(10, 6))
    calculate_state_values(method='MC')
    plt.title("Monte Carlo Estimates")
    plt.savefig('Monte_Carlo_Estimates.png', dpi=300, bbox_inches='tight')
    plt.close()

    # Second plot: TD(0) Estimates
    plt.figure(figsize=(10, 6))
    calculate_state_values(method='TD')
    plt.title("TD(0) Estimates")
    plt.savefig('TD(0)_Estimates.png', dpi=300, bbox_inches='tight')
    plt.close()

    # Third plot: Comparison of Mean Absolute Errors
    plt.figure(figsize=(10, 6))
    calculate_mean_abs_error()
    plt.title("Comparison of MC vs TD(0) Absolute Errors")
    plt.savefig('MC_vs_TD_abs_errors.png', dpi=300, bbox_inches='tight')
    plt.close()

if __name__ == '__main__':
    plot_figures()
```

B Reinforcement Learning: Group Report

B.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.

B.2 Understanding Discounted Reward MDPs

B.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 settings, 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.

B.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 B.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.

B.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:

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

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.

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:

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

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

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:

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

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.

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.

B.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 (B.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).

B.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.

B.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.

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:

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

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.

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:

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

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.

B.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.

B.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 $v_\pi$**
**Input:** the policy $\pi$ 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 $\pi$ 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
```

B.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.

B.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.

B.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.

B.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 B.2 (SARSA Update rule)**.** SARSA update rule is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \qquad \text{(B.2)}$$

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.

B.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 B.3 (Q-Learning)**.** Q-learning update rule is defined as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right], \qquad \text{(B.3)}$$

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.

B.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 B.4 (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

$$A_t = argmax_a Q(S_t, a), \tag{B.4}$$

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 B.5 (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.

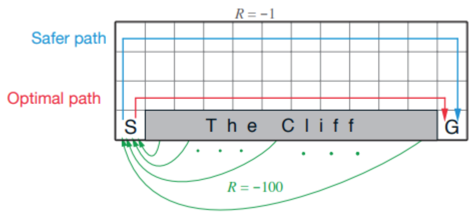

Figure B.1: Grid world for the Cliff walking example[5].

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.

B.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 B.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 B.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 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

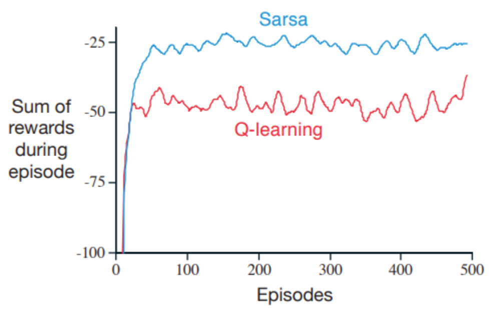

Figure B.2: Comparison between the performance of SARSA and Q-Learning[5].

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.

B.4.6 Double Q-learning

B.4.7 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.

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 B.6. (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,

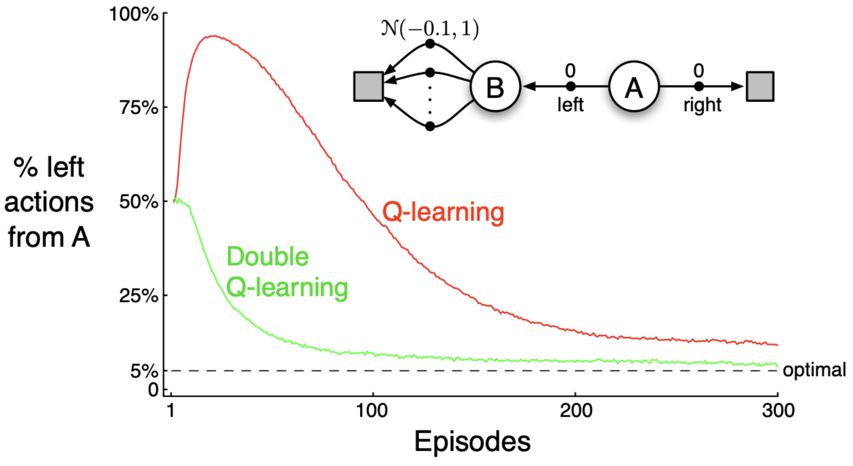

Figure B.3: Comparison of Q-learning and Double Q-learning on Example B.6. (Figure 6.5 in [5])

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 B.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.

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 B.3 clearly shows that Double Q-Learning performs much better than Q-learning in the case of Example B.6. 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, $\epsilon = 0.1$.

B.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.