Open Source Python Package Implementation of Reinforcement Learning Algorithms for Solving Markov Decision Process

Papan Yongmalwong

papan.yo594@cbs.chula.ac.th

ST62-10

Department of Statistics

Faculty of Commerce and Accountancy

Chulalongkorn University

Abstract

In this paper we aim to implement a reproducible tool for educational purposes in running reinforcement learning experiments. For simplicity, we pick an inventory control problem called *retailing skis* (Denardo 1982), formulate the problem as Markov decision process (MDP), and solve for its exact solution using dynamic programming. The exact solution obtained serves as a benchmark to assess the quality of reinforcement learning (RL) algorithms in terms of root mean squared error of action-value function, state-value function, and action following the underlying policy. We cover five fundamental RL algorithms including four tabular methods (1) Monte Carlo (2) Sarsa (3) Q-learning (4) Double Q-learning, and one value approximation

method (5) episodic semi-gradient Sarsa. We find that given enough exploration at the early time stage by following decaying epsilon greedy policy, episodic semi-gradient Sarsa, a value approximation method, outperforms the rest tabular methods in terms of root mean squared error of action-value function, state-value function, and action following the underlying policy despite a noisier manner.

Keywords: Markov Decision Process, Reinforcement Learning, Decaying Epsilon Greedy

1 Introduction

Markov decision process (MDP) is a mathematical framework for modeling environment with uncertainty where sequential decision making takes part in. Modeling environment as MDP requires the knowledge of the uncertainty and the reward function after interacting with the environment. Solving for optimal decision making is fulfilled by dynamic programming (DP). DP provides exact solution at great computational expense. DP suffers from environments with large state or action space and its exact solution becomes infeasible.

Reinforcement learning (RL) is an area of developing alternative algorithms that attempt to obtain good approximate of the exact solution to the same problem at lower computational expense and without the knowledge of the uncertainty and the reward function. Therefore, RL algorithms are sometimes called model-free algorithms.

There are recent advances in developing state-of-the-art RL algorithms to learn a good decision-making rule or a policy from computer games. Successfully applied to playing Atari 2600 computer games by DeepMind Technologies, the RL algorithm developed in which they refer to as Deep Q-Networks (DQN) achieves better performance than an expert human player on three games and it achieves close to human performance on one game with no adjustment of the architecture or hyperparameters (DeepMind Technologies 2013). DQN is a variant of Q-learning with the approximation of action-value function (Q-function) as a convolutional neural

network with 3 hidden layers. Three years from that, DeepMind Technologies of different authors proposed Double DQN, remarkably outperforming its predecessor DQN on several games (Google DeepMind 2016). Double DQN is the RL algorithm with the same architecture that accounted for overestimation bias of the Q-function. Due to a number of advances including Playing Atari with DQN and the improvement in playing games with Double DQN, RL is a very interesting area to be explored here and now.

2 Problem Formulation

Our chosen problem for this paper is *retailing skis*. The situation here is that there is a wholesaler's truck in front of a ski equipment shop at the beginning of each month to deliver pairs of skis upon request. The ski shop owner decides how many to order at the beginning of each month. Customer demand for ski during the month from the ski shop is Poisson distributed with a known rate. Future inventory is only determined from the information of current inventory. Customers pay at the end of each month. The owner borrows fund from a bank to buy skis and is needed to pay interest at the end of each month. The sequential decision to be made has an ending period.

As can be seen, the decision maker here is a ski shop owner. The quantity of skis to be purchased at the beginning of each month is under control of the owner. The uncertainty for this problem is the customer demand during the month with a known rate. The reward function for this problem can be calculated from the cost and the revenue of known functions. Moreover, the context of the problem where future inventory is only determined by the current inventory satisfies Markov property. Given that the problem is a known environment, satisfies the Markov property, and has a known reward function, the problem can be formulated as MDP.

From the context of the problem, it is straightforward that the inventory level of skis and the quantity the owner decides to purchase at the beginning of each month, together with the demand for ski during the month results in a change of the inventory level at the next time stage. It is self-implied that, in a business sense, the owner seeks maximum profit or minimum cost within the horizon. If the owner purchases more or less than the demand, an opportunity cost is incurred. One way to formulate the problem is to find a decision-making rule or a policy

which minimizes the opportunity cost. The other way around is to find a policy which maximizes the reward within the horizon. With the latter choice, the objective for *retailing skis* can be formulated as follows:

$$V(S_t) = \max_{A_t} \mathbb{E}_\pi[R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \cdots + \gamma^{T-t-1} \cdot R_T | S_t, A_t]$$

$$t \in [T_0, T-1]$$

The objective $V(.)$ is also called the value function of some state or the state-value function. Under this problem, $V(.)$ is the expected cumulative discounted profit over the remaining months. $\mathbb{E}[.]$ is a notation for taking expectation. The reward function R_{t+1} is a function of state and action at time t. Under this problem, the reward is the profit during the month and is to be discussed further. Each reward or profit is also a random variable due to the demand for ski during the month. Policy π is a notation for some decision-making rule. Therefore, $\mathbb{E}_\pi[.]$ is a notation for taking expectation under some underlying policy. State S_t is the inventory level at the beginning of month. Decision variable or action A_t is the quantity of skis to purchase at the beginning of month. The initial time T_0 is the time the business starts. The terminal time T is the time the business is close. The formulation assumes constant discount factor γ. Different policy results in different objective value. The optimal policy is the policy that results in the highest value of a state. Due to the principle of optimality, the value of a state under an optimal policy is equal to the value from taking the action that yields the highest expected return from that state. The relationship can be re-expressed as follows:

$$V^*(S_t) = \max_{A_t} \mathbb{E}[R_{t+1} + \gamma \cdot V^*(S_{t+1}) | S_t, A_t]$$

$$t \in [T_0, T-1]$$

The above equation is the famous Bellman equation and is the objective for MDP. $V^*(.)$ is a notation for optimal value function. As can be noticed, the optimal policy π^* is omitted for simplicity. Interestingly, $\mathbb{E}[R_{t+1} + \gamma \cdot V^*(S_{t+1}) | S_t, A_t]$ can be restated as the optimal Q-value $Q^*(S_t, A_t)$ or optimal action-value function as well.

The owner only decides sequentially from the beginning of the first month (time T_0) to the beginning of the last month (time $T-1$). At the end of the last month or the terminal time (time T), the business is close, and the owner is left with skis to be sold at Sale price per ski. The terminal reward or the profit at the terminal time can then be formulated as follows:

$$V(S_T) = \text{Sale} \cdot S_T$$

What is left for *retailing skis* environment to be rigorously formulated as MDP is the reward function. Consider the reward function at time t as follows:

$$R_{t+1} = \gamma \cdot \text{Retail} \cdot \min(\text{Demand}_t, S_t + A_t) - \text{Wholesale} \cdot A_t - \gamma \cdot r \cdot \text{Wholesale} \cdot (S_t + A_t)$$

The reason the reward at time t is subscripted with $t+1$ is that we consider the reward to be the profit received at the end of the month discounted back to the beginning of the month. The profit composes of the revenue $\gamma \cdot \text{Retail} \cdot \min(\text{Demand}_t, S_t + A_t)$ and the cost $\text{Wholesale} \cdot A_t + \gamma \cdot r \cdot \text{Wholesale} \cdot (S_t + A_t)$. The revenue is derived from the number of skis sold at Retail price per ski. For simplicity, it is assumed that every transaction occurred at the end of the month. The cost is derived from the number of skis purchased at Wholesale price per ski at the beginning of the month plus the interest for total skis in stock during the month. The interest rate r is assumed constant. The next inventory level or the next state can be determined as follows:

$$S_{t+1} = S_t + A_t - \text{Demand}_t$$

$$\text{Demand}_t \sim \text{Poisson}(\lambda)$$

Solving this recursively from time $T-1$ to time T_0 yields the optimal action A_t^* for every possible state at time $t \in [T_0, T-1]$. The above mathematical formulation of *retailing skis* environment is called MDP and solving for optimal action at each time in a backward recursion manner is called dynamic programming (DP). The solution obtained from DP is an exact solution. It is worth noting that one of the key reasons the exact solution is feasible with DP is due to the formulation of the environment where recombining nodes exist. The nodes here are abstract representation for inventory level at each time stage.

3 Background

In this section, we introduce pseudocodes of related concepts used in this paper. The concepts include algorithms that tackle problem where its environment has been formulated as MDP, and policies that address the exploration and exploitation dilemma. We believe that looking at pseudocodes is the most concise way of understanding the underlying concepts throughout this paper.

3.1 Dynamic Programming

To recap, we seek to solve for optimal action A_t^* which satisfies $V^*(S_t) = \max_{A_t} Q^*(S_t, A_t)$ for $t \in [T_0, T-1]$. The calculation of the expected return or the optimal Q-value at some state and action requires knowledge of the environment. In *retailing skis*, the knowledge required is the distribution of demand during the month. The function $\text{OptimalV}(S_t)$ defined in **Algorithm 1**: **Dynamic Programming** solves for optimal value at every state recursively until a stopping criterion is reached. The stopping criterion here is when the time of the state reaches the terminal time of the problem. We finally call the function at initial state in order to obtain the exact solution including optimal Q-value at every possible state and action, optimal value or V-value at every possible state, and optimal action or A-value at every possible state. The optimal policy is to take action as in the optimal action calculated beforehand by dynamic programming. Apart from this, the time complexity of DP is improved by memoization during the implementation.

Below shows the pseudocode of the algorithm.

Algorithm 1: **Dynamic Programming**

function $\text{OptimalV}(S_t)$
    if $t <$ terminal time **then**
        for each action A_i from n possible actions of state S_t **do**
            $Q^*(S_t, A_i) =$ expectation of reward $+ \gamma \cdot V(t + 1, S_{t+1})$
        end for
        return $\max_{A_i} Q^*(S_t, A_i)\ \forall i = 1,2, \dots, n$
    end if
    if $t ==$ terminal time **then**
        return terminal reward at state S_t
    end if
end function
Call $\text{OptimalV}(S_{\text{intial time}})$

3.2 Monte Carlo Method

We introduce the first RL algorithm covered in this paper called Monte Carlo method. Monte Carlo method in this context is a way of solving MDP problem based on averaging sample returns. Sample return is the sum of discounted actual return observed by decision maker. In *retailing skis*, the owner decides quantity of skis to purchase at the beginning of month and observes actual reward at the end of month from the first month to the last month of business in order to obtain a sample return G. Mathematically, sample return at time t is defined as $G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \cdots + \gamma^{T-t-1} \cdot R_T$. By repeating this indefinitely, the owner experiences decision making many times and if some certain rule is applied appropriately, the owner can make good decision by time. That certain rule or the policy is to approximate expected return $Q(S_t, A_t)$ or the Q-value with average sample returns $\hat{Q}(S_t, A_t) = \frac{1}{N(S_t, A_t)} \sum_{i=1}^{N(S_t, A_t)} G_i$ for number of visits $N(S_t, A_t)$ and make decision by taking random action most of the time in the early stage and by taking the action that yields the highest average sample returns from that state most of the time in later stage or once we are sure of the Q-value estimate. That is, we approximate optimal action $\hat{A}_t^*$ which satisfies $\hat{V}^*(S_t) = \max_{A_t} \hat{Q}(S_t, A_t)$ for $t \in [T_0, T-1]$ in the later stage. With this certain rule, the solution can be achieved without visiting every possible outcome like dynamic programming, making it feasible in large state space MDP. As can be noticed, it is required for Monte Carlo method to store $\hat{Q}(.,.)$ on a table for making comparison in the established rule. Therefore, Monte Carlo is considered a tabular method.

In the context of *retailing skis*, the owner decides the quantity of skis to purchase with respect to her policy at the beginning of the month, takes notes of inventory level and the decision made at the beginning of the month. After 1 month, the owner takes notes of the reward incurred during the month and the next inventory level after 1 month. The owner repeats this and takes notes sequentially until reaching the terminal time. At terminal time, the owner updates her knowledge back from the end to the start. By updating her knowledge means to update Q-value at a visited state-action pair with the average of actual returns backwards in time.

By law of large number, $\hat{Q}(.,.)$ approaches the Q-value $Q_\pi(.,.)$ following some underlying fixed policy π as number of visits to each state-action pair approaches infinity (Richard S. Sutton 2017). However, the policy is not fixed but rather adapts dynamically as mentioned above while learning from interacting with the

environment. With this dynamic, the policy is expected to move towards optimal policy and $\hat{Q}(.,.)$ moves closer to the optimal Q-value $Q^*(.,.)$ over time.

Observing actual rewards from time to time in real world can be time-consuming or sometimes impractical. Another way to observe rewards in no time is to simulate the rewards from a known environment. This way, the learning can be achieved in no time. It has to be noted that Monte Carlo method itself requires no prior knowledge of the environment in order to approximate the solution. But with some prior knowledge, we can speed up the learning without observing actual rewards.

The term episode described in **Algorithm 2: Monte Carlo** is another word for one period where a decision maker interacts with the environment sequentially from initial time to terminal time. The learning is conducted indefinitely as the number of episodes approach infinity. The average sample returns $\hat{Q}(S_t, A_t) = \frac{1}{N(S_t,A_t)} \cdot \sum_{i=1}^{N(S_t,A_t)} G_i$ can be restated as an update rule as follows:

$$\hat{Q}(S_t, A_t) \leftarrow \hat{Q}(S_t, A_t) + \frac{1}{N(S_t, A_t)} \cdot (G - \hat{Q}(S_t, A_t))$$

To initialize the environment, it is required to implement reward function for the decision maker to interact with by inputs including current state and current action and observe the results including reward and the next state. Additionally, it is required to implement terminal reward function for handling the reward at terminal time with inputs including current state as well. We finally call the function fit with inputs including the environment ENV and number of episodes EPISODES to obtain approximate solution to MDP problem.

Below is the pseudocode of the algorithm.

Algorithm 2: Monte Carlo

---

function fit(ENV, EPISODES)

Initialize $\hat{Q}(.,.) = 0$ and $N(.,.) = 0$ $\forall$state, action

Initialize some underlying policy π, e. g. Epsilon Greedy

for episode $= 1$ to EPISODES **do**

Let state $=$ initial state, action $=$ get action with input
$= \hat{Q}$(initial state, .) from π

for t from inital time to terminal time -1 **do**

Store state to S_t, store action to A_t

Interact with ENV to get reward and next state with input $=$ state and action

Store reward to R_{t+1}

if $t+1 <$ terminal time **then**

Let next action $=$ get action with input $= \hat{Q}$(next state, .) from π

state $\leftarrow$ next state, action $\leftarrow$ next action

else

state $\leftarrow$ next state

end if

end for

Initialize actual cumulative discounted reward $G = 0$

for t from terminal time to initial time **do**

if $t ==$ terminal time **then**

$G \leftarrow G +$ terminal reward at state S_t

else

$G \leftarrow G + \gamma \cdot R_{t+1}$

end if

$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$

$\hat{Q}(S_t, A_t) \leftarrow \hat{Q}(S_t, A_t) + \frac{1}{N(S_t, A_t)} \cdot (G - \hat{Q}(S_t, A_t))$

end for

end for

end function

Initialize environment ENV and number of episodes EPISODES

Call fit(ENV, EPISODES)

3.3 Sarsa

The second RL algorithm to be discussed is called Sarsa. Like Monte Carlo method, Sarsa is a tabular method because it requires storing $\hat{Q}(.,.)$ on a table. The difference between Monte Carlo and Sarsa is that Monte Carlo interacts with the environment sequentially from the initial time to the terminal time to obtain full sequence of $\dots, S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \dots$ beforehand and update $\hat{Q}(.,.)$ at the visited state-action pair from the terminal time back to the initial time while Sarsa interacts with the environment, observes immediate reward R_{t+1}, takes the next action A_{t+1} by following the underlying policy, and updates $\hat{Q}(.,.)$ at state S_t and action A_t with the quintuple $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$, and repeat until reaching terminal time. Mathematically, the update rule for Sarsa is as follows:

$$\hat{Q}(S_t, A_t) \leftarrow \hat{Q}(S_t, A_t) + \mathrm{lr} \cdot (R_{t+1} + \gamma \cdot \hat{Q}(S_{t+1}, A_{t+1}) - \hat{Q}(S_t, A_t))$$

for some learning rate $\mathrm{lr} \in (0,1)$. As can be seen, the algorithm name stemmed from the fact that its update rule consists of the quintuple $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$, the sequence of current state, current action, immediate reward, next state, and next action at time t. Moreover, the update rule is learning a guess from a guess. This is sometimes referred to as bootstrapping. The term $R_{t+1} + \gamma \cdot \hat{Q}(S_{t+1}, A_{t+1})$ is often referred to as the bootstrapping target.

In the context of *retailing skis*, the owner decides to purchase a quantity of skis with respect to her policy at the beginning of the month and observes the reward incurred during the month. At the end of the month or the beginning of next month, the owner decides to purchase a quantity of skis with respect to her policy for demand to be incurred next month. The owner then updates the knowledge (Q-value) of arriving at this state and of taking this action at the beginning of the month by an increment proportional to the immediate reward plus discounted Q-value of arriving at the next state and of taking the next action at the beginning of next month less current Q-value. As can be noticed, the owner only waits 1 month to update her knowledge with Sarsa update rule.

It is important to point out that the update rule of Sarsa at current state-action pair requires information from the next state-action pair. Once the Q-value at current state-action pair is updated, the decision maker traverses to

that next state-action pair and repeats for updates until reaching terminal time. This is called on-policy because the decision maker follows that next state-action pair right after current update.

Sarsa is proven to converge to optimal policy and optimal Q-value as number of visits approaches infinity and the dynamic of the policy approaches pure exploitation (Singh 2000). The dynamic of policy described here as a condition for Sarsa to converge is the same as we have described in Monte Carlo method, to explore most of the time in early stage and to exploit most of the time in later stage.

The pseudocode of the algorithm is shown in below. It can be clearly seen that for each episode, Sarsa learns forwards in time, as opposed to Monte Carlo method.

Algorithm 3: **Sarsa**

---

function fit(ENV, EPISODES)

Initialize $\hat{Q}(.,.) = 0 \; \forall \text{state}, \text{action}$

Initialize some underlying policy π, e. g. Epsilon Greedy

Initialize some learning rate $\text{lr} \in (0,1)$

for episode $= 1$ to EPISODES **do**

Let state $=$ initial state, action $=$ get action with input $= \hat{Q}(\text{initial state}, .)$ from π

for t from inital time to terminal time $- 1$ **do**

Interact with ENV to get reward and next state with input $=$ state and action

if $t + 1 <$ terminal time **then**

Let next action $=$ get action with input $= \hat{Q}(\text{next state}, .)$ from π

Let target $U = \text{reward} + \gamma \cdot \hat{Q}(\text{next state}, \text{next action})$

else

Let target $U = \text{reward} + \gamma \cdot$ terminal reward at next state

end if

$\hat{Q}(\text{state}, \text{action}) \leftarrow \hat{Q}(\text{state}, \text{action}) + \text{lr} \cdot (U - \hat{Q}(\text{state}, \text{action}))$

state ← next state, action ← next action

end for

end for

end function

Initialize environment ENV and number of episodes EPISODES

Call fit(ENV, EPISODES)

3.4 Q-learning

Arriving at the third RL algorithm, Q-learning is another tabular method similar to Sarsa. Contrary to updating current Q-value with the quintuple, Q-learning updates current Q-value with the quadruple $S_t, A_t, R_{t+1}, S_{t+1}$. Information of the next action is not required to update Q-value in Q-learning because it chooses the next action greedily based on the Q-value at the next state. Mathematically, the update rule for Q-learning is as follows:

$$\hat{Q}(S_t, A_t) \leftarrow \hat{Q}(S_t, A_t) + \text{lr} \cdot (R_{t+1} + \gamma \cdot \max_{A_{t+1}} \hat{Q}(S_{t+1}, A_{t+1}) - \hat{Q}(S_t, A_t))$$

for some learning rate $\text{lr} \in (0,1)$. The bootstrapping target here is $R_{t+1} + \gamma \cdot \max_{A_{t+1}} \hat{Q}(S_{t+1}, A_{t+1})$. Selecting the next action greedily results in the overestimation of corresponding Q-value and V-value. Thereby the use of Q-learning introduces a problem called maximization bias.

In the context of *retailing skis*, the owner decides to purchase a quantity of skis with respect to her policy at the beginning of the month and observes the reward incurred during the month. At the end of the month or the beginning of next month, the owner updates the knowledge (Q-value) of arriving at the state-action pair at the beginning of last month by an increment proportional to the reward incurred during the month plus discounted Q-value of arriving at the next state-action pair where the action is taken greedily less Q-value of the last month state-action pair. Similar to Sarsa, the owner only waits 1 month to update her knowledge with Q-learning update rule.

It is important to point out that Q-learning does not follow the greedy policy used to obtain the next action during the update. After the update, the next action is drawn from the policy independent of the last update to obtain the current state-action pair. Therefore, the algorithm is considered off-policy.

The pseudocode of the algorithm is shown in below. It can be clearly seen that for each episode, Q-learning learns forwards in time, similar to Sarsa.

Algorithm 4: **Q − learning**

function fit(ENV, EPISODES)

Initialize $\hat{Q}(.,.) = 0$ ∀state, action

Initialize some underlying policy π, e. g. Epsilon Greedy and a greedy policy π_{greedy}

Initialize some learning rate lr ∈ (0,1)

for episode = 1 to EPISODES **do**

Let state = initial state

for t from inital time to terminal time − 1 **do**

Let action = get action at state from π

Interact with ENV to get reward and next state with input
= state and action

if $t + 1 <$ terminal time **then**

Let next action = get action with input
= $\hat{Q}$(next state, .) from π_{greedy}

Let target U = reward + $\gamma \cdot Q$(next state, next action)

else

Let target U = reward + $\gamma \cdot$ terminal reward at next state

end if

$\hat{Q}$(state, action) ← $\hat{Q}$(state, action) + lr · ($U - \hat{Q}$(state, action))

state ← next state

end for

end for

end function

Initialize environment ENV and number of episodes EPISODES

Call fit(ENV, EPISODES)

3.5 Double Q-learning

The fourth RL algorithm is a variant of Q-learning called double Q-learning developed to address the maximization bias in Q-learning. Interestingly, we can overcome the problem by initializing two separate Q-value tables $\hat{Q}_1(.,.)$ and $\hat{Q}_2(.,.)$, and follow an update rule where one of the Q-value is updated by taking greedy action based on another Q-value with probability 0.5, and vice versa. This removes the systematic bias during each update. Mathematically, the update rule for double Q-learning can be stated as follows:

$$\hat{Q}_1(S_t, A_t) \leftarrow \hat{Q}_1(S_t, A_t) + \text{lr} \cdot \left(R_{t+1} + \gamma \cdot \max_{A_{t+1}} \hat{Q}_2(S_{t+1}, A_{t+1}) - \hat{Q}_1(S_t, A_t) \right) \text{w. p. } 0.5$$

for some learning rate $\text{lr} \in (0,1)$, and vice versa. The bootstrapping target here for updating $\hat{Q}_1(S_t, A_t)$ is $R_{t+1} + \gamma \cdot \max_{A_{t+1}} \hat{Q}_2(S_{t+1}, A_{t+1})$, and the other way around.

The pseudocode of the algorithm is shown in below.

Algorithm 5: **Double Q − learning**

function fit(ENV, EPISODES)

Initialize $\hat{Q}_1(.,.) = 0$ and $\hat{Q}_2(.,.) = 0$ $\forall$state, action

Initialize some underlying policy π, e. g. Epsilon Greedy and a greedy policy π_{greedy}

Initialize some learning rate $\text{lr} \in (0,1)$

for episode $= 1$ to EPISODES **do**

Let state $=$ initial state

for t from inital time to terminal time $- 1$ **do**

Let action $=$ get action at state from π

Interact with ENV to get reward and next state with input $=$ state and action

if rand$(0,1) < .5$ **then**

if $t + 1 <$ terminal time **then**

Let next action $=$ get action with input $= \hat{Q}_1(\text{next state}, .)$ from π_{greedy}

Let target $U = \text{reward} + \gamma \cdot \hat{Q}_2(\text{next state}, \text{next action})$

else

Let target U = reward + γ · terminal reward at next state

end if

$\hat{Q}_1(\text{state}, \text{action}) \leftarrow \hat{Q}_1(\text{state}, \text{action}) + \text{lr} \cdot (U - \hat{Q}_1(\text{state}, \text{action}))$

else

if $t + 1 <$ terminal time **then**

Let next action = get action with input $= \hat{Q}_2(\text{next state}, .)$ from π_{greedy}

Let target U = reward + $\gamma \cdot \hat{Q}_1(\text{next state}, \text{next action})$

else

Let target U = reward + γ · terminal reward at next state

end if

$\hat{Q}_2(\text{state}, \text{action}) \leftarrow \hat{Q}_2(\text{state}, \text{action}) + \text{lr} \cdot (U - \hat{Q}_2(\text{state}, \text{action}))$

state ← next state

end for

end for

end function

Initialize environment ENV and number of episodes EPISODES

Call fit(ENV, EPISODES)

3.6 Episodic Semi-gradient Sarsa (Linear Approximation)

This is the last RL algorithm and is considered the only value approximation method introduced in this paper. Unlike tabular method, value approximation method incorporates a differentiable real-valued function of state-action pair S_t, A_t and some parameters or weights $\mathbf{W}$ in place of a table in representing the Q-value. Mathematically, the function $\hat{Q}(S_t, A_t; \mathbf{W}): \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d \rightarrow \mathbb{R}$ serves as an approximate function to the

tabular Q-values $\hat{Q}(S_t, A_t)$. Value approximation method allows less memory because it only stores the value of weights $\mathbf{W}$ of size much less than the size of Q-value $\hat{Q}(.,.)$ on table. Moreover, it allows generalization to unseen state-action pairs and potentially converge to optimal Q-value faster if the function $\hat{Q}(.,.:\mathbf{W})$ is chosen appropriately to the problem.

The update rule in value approximation method is put differently. Consider the gradient descent update rule of weights $\mathbf{W}$ defined as $\mathbf{W} \leftarrow \mathbf{W} - \frac{1}{2} \cdot \text{lr} \cdot \nabla \left(Q^*(S_t, A_t) - \hat{Q}(S_t, A_t: \mathbf{W})\right)^2$ for some learning rate $\text{lr} \in (0,1)$. The update moves $\mathbf{W}$ to the steepest descent direction of function $\left(Q^*(S_t, A_t) - \hat{Q}(S_t, A_t: \mathbf{W})\right)^2$. Intuitively, we seek $\mathbf{W}$ that minimizes the squared error of the optimal Q-value $Q^*(S_t, A_t)$ and the approximate Q-function $\hat{Q}(S_t, A_t: \mathbf{W})$ with a first-order iterative optimization method. The gradient descent update rule can be restated as The update rule can be restated as $\mathbf{W} \leftarrow \mathbf{W} + \text{lr} \cdot \left(Q^*(S_t, A_t) - \hat{Q}(S_t, A_t: \mathbf{W})\right) \nabla \hat{Q}(S_t, A_t, \mathbf{W})$. But the optimal Q-value $Q^*(S_t, A_t)$ or the true target is not known and is in fact the value we try to get with RL algorithm. Hence a bootstrapping target U_t is used in place of the true target. The examples of bootstrapping target are what we have discussed in the update rule of Sarsa, Q-learning, and double Q-learning. We arrive at what is referred to as semi-gradient method with the update rule $\mathbf{W} \leftarrow \mathbf{W} + \text{lr} \cdot \left(U_t - \hat{Q}(S_t, A_t: \mathbf{W})\right) \nabla \hat{Q}(S_t, A_t, \mathbf{W})$. Although Semi-gradient method does not converge robustly as gradient descent, it shows remarkable performance in many RL experiments, notably DQN (DeepMind Technologies 2013) and double DQN (Google DeepMind 2016).

Semi-gradient extends its way to Sarsa straightforwardly. The bootstrapping target in what we called episodic semi-gradient Sarsa is defined as $U_t = R_{t+1} + \gamma \cdot \hat{Q}(S_{t+1}, A_{t+1}: \mathbf{W})$. The policy dynamic and the on-policy learning from Sarsa apply to the episodic semi-gradient Sarsa as well.

In this paper we consider episodic semi-gradient Sarsa with $\hat{Q}(.,.:\mathbf{W})$ as a linear function defined as $\hat{Q}(S_t, A_t: \mathbf{W}) = \mathbf{X}_{S_t, A_t}^T \mathbf{W}$ for some feature $\mathbf{X}_{S_t, A_t}$ of the same size as $\mathbf{W}$. Feature $\mathbf{X}_{S_t, A_t}$ can be viewed as the transformation of state-action pair value into a vector of higher dimension allowing a better fit of the approximate function to the target. The update rule for episodic semi-gradient Sarsa with linear approximation can then be derived as follows:

$$\mathbf{W} \leftarrow \mathbf{W} + \text{lr} \cdot \left(R_{t+1} + \gamma \cdot \mathbf{X}_{S_{t+1}, A_{t+1}}^T \mathbf{W} - \mathbf{X}_{S_t, A_t}^T \mathbf{W}\right) \cdot \mathbf{X}_{S_t, A_t}$$

for some learning rate $\text{lr} \in (0,1)$. The bootstrapping target here is $R_{t+1} + \gamma \cdot \mathbf{X}_{S_{t+1},A_{t+1}}^{T}\mathbf{W}$.

Below is the pseudocode for the algorithm discussed.

Algorithm 6: **Episodic Semi − gradient Sarsa** (**Linear Approximation**)

function fit(ENV, EPISODES)

Intiatialize a rule for constructing feature $\mathbf{X}_{\text{state,action}} \in \mathbb{R}^d$ from arbitrary state and action

Initialize weight $\mathbf{W} = \mathbf{0} \in \mathbb{R}^d$

Define a linear function $\hat{Q}(\text{state, action: } \mathbf{W}) = \mathbf{X}_{\text{state,action}}^{T}\mathbf{W}$

Initialize some underlying policy π, e. g. Epsilon Greedy

Initialize some learning rate $\text{lr} \in (0,1)$

for episode $= 1$ to EPISODES **do**

Let state $=$ initial state, action $=$ get action with input $= \hat{Q}(\text{state}, ., \mathbf{W})$ from π

for t from inital time to terminal time $- 1$ **do**

Interact with ENV to get reward and next state with input $=$ state and action

if $t + 1 <$ terminal time **then**

Let next action $=$ get action with input $= \hat{Q}(\text{next state}, .: \mathbf{W})$ from π

Let target $U = \text{reward} + \gamma \cdot \hat{Q}(\text{next state, next action: } \mathbf{W})$

else

Let target $U = \text{reward} + \gamma \cdot$ terminal reward at next state

$\mathbf{W} \leftarrow \mathbf{W} + \text{lr} \cdot \left(U - \hat{Q}(\text{state, action: } \mathbf{W})\right) \cdot \mathbf{X}_{\text{state,action}}$

end if

state $\leftarrow$ next state, action $\leftarrow$ next action

end for

end for

end function

Initialize environment ENV and number of episodes EPISODES

Call fit(ENV, EPISODES)

3.7 Epsilon Greedy

Epsilon greedy is a simple decision-making rule or policy that balances between the exploration and exploitation during the learning of RL algorithm. Generally, we keep a small fraction of epsilon as probability of the policy to explore – to choose action from a set of possible actions in a state randomly and the rest of the fraction is the probability of the policy to exploit our current knowledge – to choose action $\hat{A}_t^*$ which satisfies $\hat{V}^* = \max_{A_t} \hat{Q}(S_t, A_t)$ for $t \in [T_0, T-1]$. Epsilon greedy is a basic policy applied to RL algorithms in **3.2** to **3.6**. Policy that explores too much can slow the RL algorithm in learning the optimal solution. At the same time, policy that exploits too much can cause the RL algorithm to fall into local optimum. This is referred to as the exploration and exploitation dilemma. Keeping a fixed epsilon often results in a poorer performance than a decaying epsilon to be discussed next.

Greedy policy is a policy of pure exploitation. As can be noticed, the greedy policy is equivalent to epsilon greedy policy where epsilon equals 0. In contrast, random policy or a policy of pure exploration is equivalent to epsilon greedy policy where epsilon equals 1.

The pseudocode of epsilon greedy policy is shown below.

Policy 1: **Epsilon Greedy**

function get_action(Q(state, .), EPSILON):

    if rand(0,1) < EPSILON **then**

        return random action from all possible actions at state

    end if

    return $\underset{\text{action}}{\text{argmax}}\, Q(\text{state}, \text{action})$

end function

3.8 Decaying Epsilon Greedy

Decaying epsilon greedy is a policy where exploration is undertaken many times in the beginning and the level of exploration decays over time, resulting in more exploiting events in later stage. Decaying epsilon greedy considered in this paper is to keep a fraction of epsilon inverse proportional to number of episodes.

The experiment conducted in this paper follows the decaying epsilon greedy as shown in the pseudocode below.

Policy 2: **Decaying Epsilon Greedy**

function get_action(Q(state, .), EPSILON, DECAY, EPISODE):

if rand(0,1) $< \frac{1}{1 + \frac{\max(\text{EPISODE} - \text{DECAY}, 0)}{\text{DECAY}}}$ **then**

return random action from all possible actions at state

end if

return $\underset{\text{action}}{\text{argmax}}\, Q(\text{state}, \text{action})$

end function

4 Python Package `ski`

We are proud to present Python package `ski`, a playground where you can run RL experiments in Python with the currently available environment *retailing skis*. The version as of the day of writing this paper is `ski 0.1.10`. The package includes six objects (1) `Environment` (2) `EpsilonGreedy` (3) `MonteCarlo` (4) `Sarsa` (5) `QLearning` (6) `DoubleQLearning`, and one utility function (7) `set_seed`.

With package ski, you can use `set_seed(num)` for running reproducible results. Next, you can use `env = Environment(sale, wholesale, retail, r, gamma, N, M, lam)` for initializing the parameters for environment *retailing skis*. Here, `sale` is the price per ski at terminal time, `wholesale` is the price per ski from the wholesaler, `retail` is the price per ski from the ski shop, `r` is the interest rate per one time interval, `gamma` is the discount factor per one time interval, `N` is the terminal time, `M` is the maximum capacity for storing ski at the ski shop during the month, and `lam` is the demand rate for ski during the month. Next, you can use `policy = EpsilonGreedy(eps, decay)` for initializing either epsilon or decaying epsilon greedy policy. Here, `eps`

is a real number between 0 (inclusive) and 1 (inclusive), and `decay` is a hyperparameter for decaying epsilon greedy. You can see pseudocode in **Policy 2: Decaying Epsilon Greedy** for more information. Next, you can use `agent =` with one of the following:

```
MonteCarlo(policy, method = 'tabular')

Sarsa(policy, lr = .1, method = 'tabular')

QLearning(policy, lr = .1, method = 'tabular')

DoubleQLearning(policy, lr = .1, method = 'tabular')

Sarsa(policy, lr = .1, method = 'approx', degree = 2)
```

for constructing an RL algorithm, also referred to as an agent. Following that, you can use `agent.fit(env, episodes)` to learn by interacting with the environment `env` sequentially until the specified number of episodes `episodes` is reached. With the same method `agent.fit`, you can use `agent.fit(env, episodes, verbose, verbose_freq, silent_plot, plot_freq)` with `verbose = True` for showing progress bar or `verbose = False` otherwise, `verbose_freq` for specifying the frequency of showing progress where input can be either a real number between 0 (inclusive) and 1 (inclusive) for showing progress once every `verbose_freq*episodes` episodes or a list of episodes for showing progress if current episode is in the list, `silent_plot = True` for silencing plot results or `silent_plot = False` otherwise, and `plot_freq` for specifying the frequency of showing plot results where input can be either a real number between 0 (inclusive) and 1 (inclusive) for showing plot results once every `plot_freq*episodes` episodes or a list of episodes for showing plot results if current episode is in the list.

Below shows an example of output after executing the following code:

```
Environment(sale = 3, wholesale = 5, retail = 7, r = .1, gamma = 1/(1+.1), N = 7, M = 20, lam = 9)

policy = EpsilonGreedy(decay = 1000)

agent = Sarsa(policy, lr = .1, method = 'approx', degree = 2)

agent.fit(env, episodes = 100000, verbose = False, plot_freq = [1000, 10000, 100000])
```

Figure 1 shows that episodic semi-gradient Sarsa Q-values move closer to the true target as number of episodes increases. As can be noticed from the figure, Q-value at time $= 1$ is always a line curve over axis state $= 0$ and axis Q-value. This is because there is only one start state which is $S_1 = 0$ (the default start state). In other

words, the initial inventory level of the ski shop at initial time $= 1$ is 0. As can be further noticed from the figure, Q-value at time $= 7$ is always an overlapping line curve over axis action $= 0$ and axis Q-value. When the business reaches the terminal time, there is no ski to be purchased further (quantity of skis to purchase at terminal time is 0). And the terminal reward is determined by only the state value at that time. Therefore, Q-value at terminal time is always deterministic.

Figure 2 shows the convergence of V-values in episodic semi-gradient Sarsa. As can be seen from the figure, the approximate optimal V-value (red) is closer to the optimal V-value (blue) as number of episodes increases. This is aligned to Figure 1 because V-value is the approximate optimal V-value and is defined as the Q-value at that state and the best action. Mathematically, V-value or the approximate optimal V-value is defined as $\hat{V}^*(S_t) = \max_{A_t} \hat{Q}(S_t, A_t)$ as we have discussed in Monte Carlo method.

Figure 3 shows the convergence of A-values in episodic semi-gradient Sarsa. As can be seen from the figure, the approximate optimal A-value (red) is closer to the optimal A-value (blue) as number of episodes increases. A-value here is defined as the approximate optimal action at a state. Mathematically, A-value or the approximate optimal action is defined as $\hat{A}^*(S_t) = \underset{A_t}{\operatorname{argmax}} \hat{Q}(S_t, A_t)$.

The results from 4 other RL algorithms are shown in **Appendix A**.

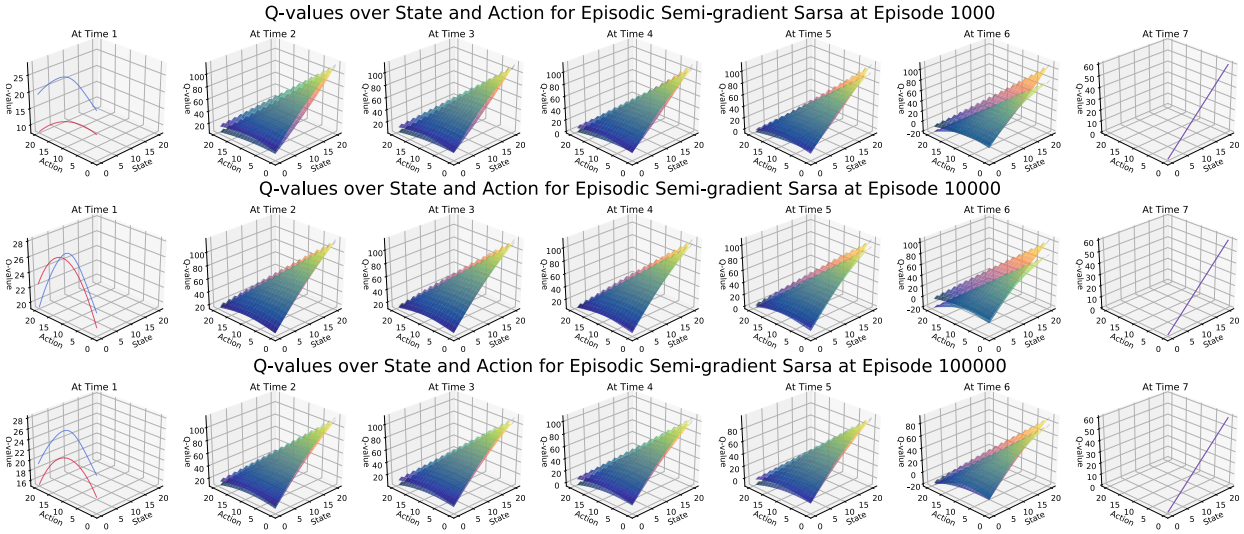

Figure 1: Episodic semi-gradient Sarsa convergence of the approximate Q-function (red) benchmarked against the optimal Q-value (blue) after 1,000 10,000 and 100,000 episodes

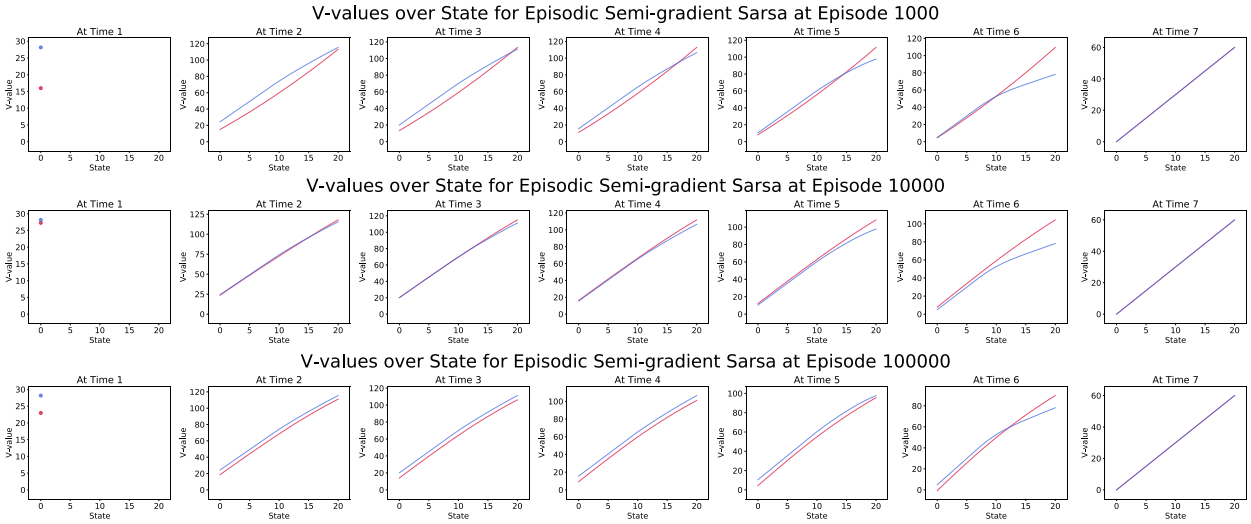

Figure 2: Episodic semi-gradient Sarsa convergence of the approximate V-function (red) benchmarked against the optimal V-value (blue) after 1,000 10,000 and 100,000 episodes

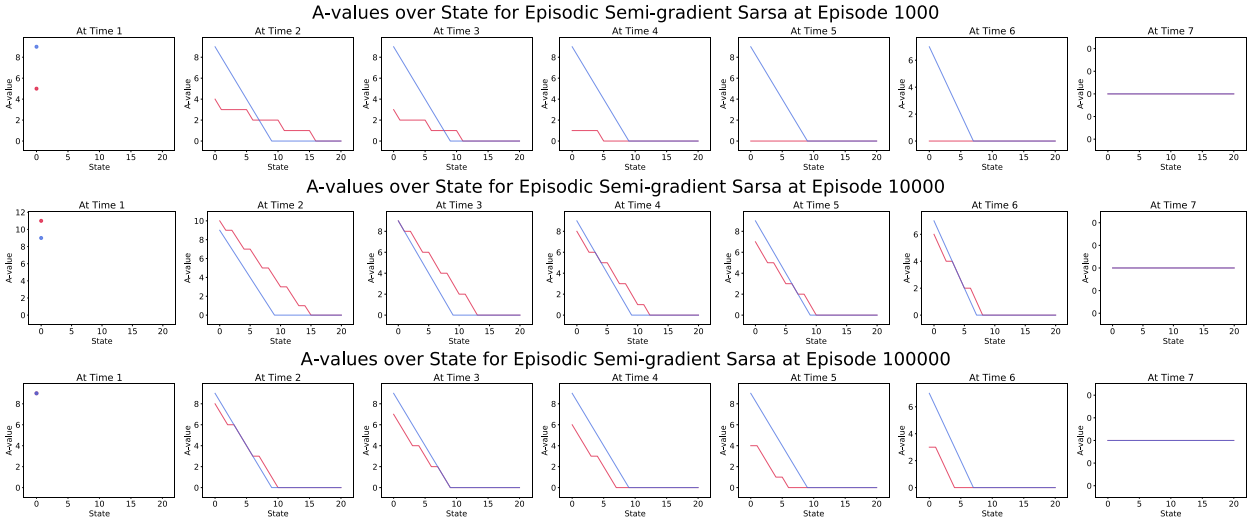

Figure 3: Episodic semi-gradient Sarsa convergence of the approximate optimal action (red) benchmarked against the optimal action A-value (blue) after 1,000 10,000 and 100,000 episodes

5 Experimental Results

To accomplish performance comparison between RL algorithms in the environment *retailing skis*, we define three metrics to assess the performance.

Root Mean Squared Error of Q-values (RMSE-Q) defined as $\frac{1}{n}\sum_{i=1}^{n}\left(Q^*(S_t, A_t) - \hat{Q}(S_t, A_t)\right)^2$ for Q-value $\hat{Q}(.,.)$ of size n. This metric is a way of assessing how close the Q-value learned at each state-action pair is to the true target.

Root Mean Squared Error of V-values (RMSE-V) defined as $\frac{1}{n}\sum_{i=1}^{n}\left(V^*(S_t) - \hat{V}(S_t)\right)^2$ for V-value $\hat{V}(.)$ of size n. This metric is a way of assessing how close the V-value learned at each state value is to the true target.

Root Mean Squared Error of A-values (RMSE-A) defined as $\frac{1}{n}\sum_{i=1}^{n}\left(A^*(S_t) - \hat{A}(S_t)\right)^2$ for A-value $\hat{A}(.)$ of size n. This metric is a way of assessing if the RL algorithm makes the right decision over time of learning.

We conduct an experiment by running 30 independent learning for each RL algorithm. Each learning of an RL algorithm interacts with the environment sequentially for $100{,}000$ episodes. The policy for each learning is the decaying epsilon policy at $\text{decay} = 1{,}000$. The parameters for the environment are as follows:

```
Environment(sale = 3, wholesale = 5, retail = 7, r = .1, gamma = 1/(1+.1), N = 7,
M = 20, lam = 9)
```

We calculate the RMSE-Q, RMSE-V, and RMSE-A once every 500 episodes to avoid high computational expense. We aggregate RMSE-Q, RMSE-V, and RMSE-A at each episode by 0.025-quantile, mean, and 0.975-quantile. The results are shown in figure 4, figure 5, and figure 6.

We want to highlight the improvement of episodic semi-gradient Sarsa over time in terms of RMSE-Q, RMSE-V, and RMSE-A. Surprisingly, as can be seen from figure 4 to 6, episodic semi-gradient Sarsa outperforms the rest of 4 tabular methods in all three metrics. From figure 4, it is sufficient to conclude that the RMSE-Q of episodic semi-gradient Sarsa at episode over 10,000 is different from the RMSE-Q of four other tabular methods at 0.05 significance level. From figure 5, it is also sufficient to conclude that the RMSE-V of episodic semi-gradient Sarsa at episode over 10,000 is different from the RMSE-Q of four other tabular methods at 0.05 significance level. On the other hand, looking at figure 6, it is not sufficient to conclude that the RMSE-A of episodic semi-gradient Sarsa for the entire range of 100,000 episodes is different from the RMSE-A of Monte Carlo method at 0.05 significance level. But it is arguably true that RMSE-A of episodic semi-gradient Sarsa is

lower than Sarsa, Q-learning, and double Q-learning in most of the episodes. To conclude, episodic semi-gradient Sarsa converges closer to the exact solution over time despite a noisier manner especially in terms of RMSE-A.

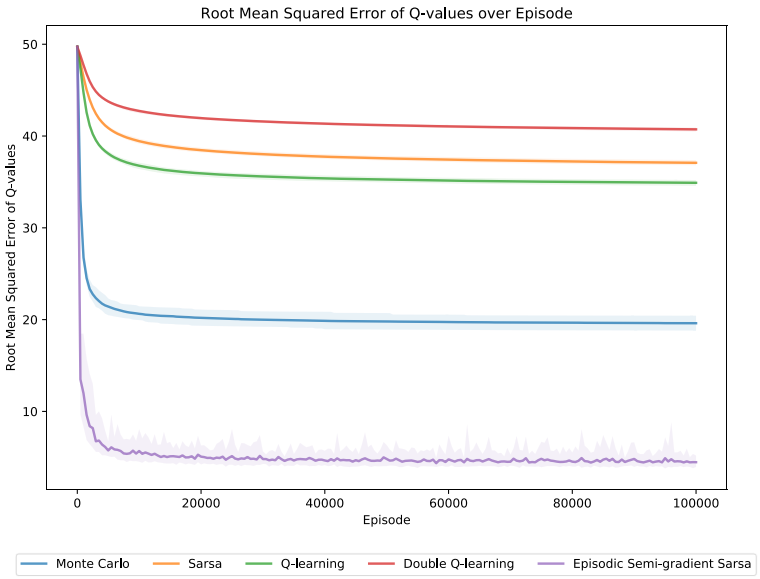

Figure 4: The plot shows the improvement of RMSE-Q over episode, the calculations are done once every 500 episodes

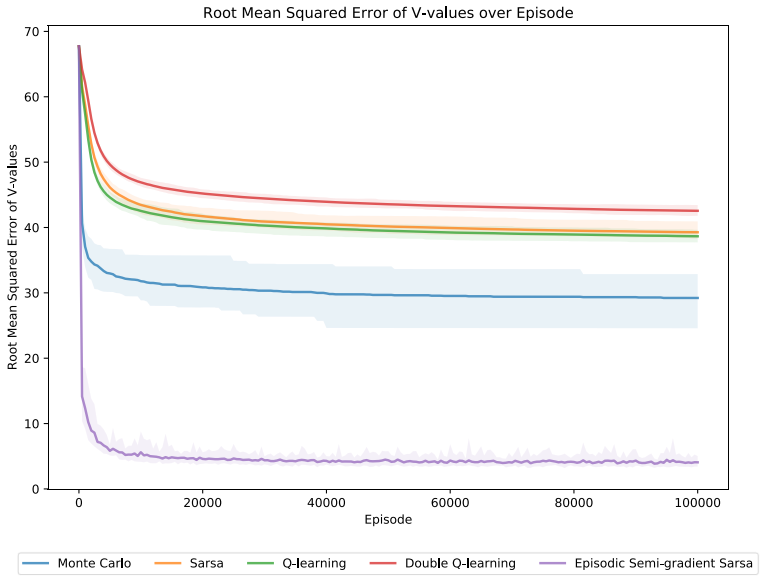

Figure 5: The plot shows the improvement of RMSE-V over episode, the calculations are done once every 500 episodes

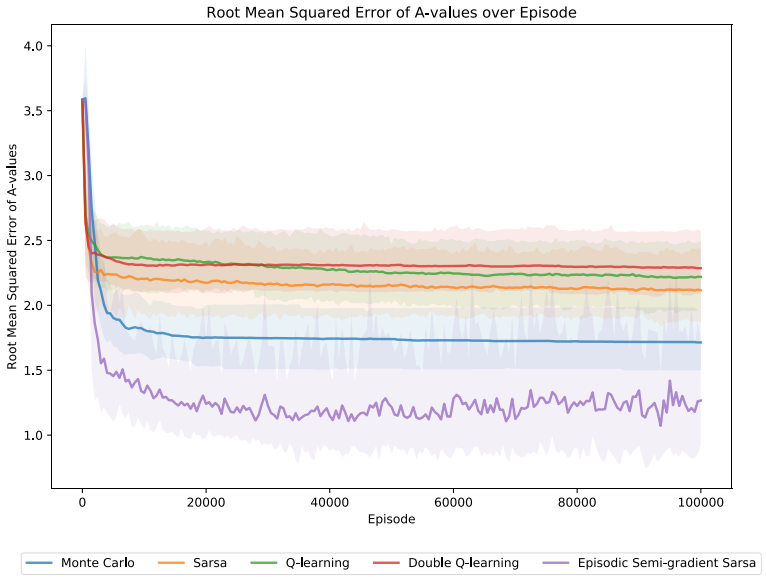

Figure 6: The plot shows the improvement of RMSE-A over episode, the calculations are done once every 500 episodes

6 Concluding Remarks

This paper has introduced a reproducible tool for running RL experiments in a setting of *retailing skis*. Although currently limited to one environment, the code is well-taken care of and is ready to serve future environments. Apart from the environment, five RL algorithms are implemented, four of which are tabular methods and the other one is a value approximation method, n-step temporal difference (TD) methods are another family of RL algorithms that span a spectrum with Monte Carlo at one end and one-step TD including Sarsa, Q-learning, and double Q-learning at the other, and have been proven better results in terms of average root mean squared error on the random walk problem (Richard S. Sutton 2017). To further improve the time complexity during the implementation, binary tree can be used in place of hash map for storing the action-value function at some state $\hat{Q}(S_t, .)$. Using binary tree for storing $\hat{Q}(S_t, .)$ of size n will improve drastically the time complexity of taking max operation over itself from $O(n)$ to $O(\log n)$, in exchange for a slight slower accessing time at each action in $\hat{Q}(S_t, .)$ during update from $O(1)$ to $O(\log n)$.

References

DeepMind Technologies. 2013. "Playing Atari with Deep Reinforcement Learning."

Denardo, Eric V. 1982. *Dynamic Programming: Models and Applications.* Dover Publications.

Google DeepMind. 2016. "Deep Reinforcement Learning with Double Q-Learning."

Richard S. Sutton, Andrew G. Barto. 2017. *Reinforcement Learning: An Introduction.* The MIT Press.

Singh, Jaakkola, Littman, Szepesvári. 2000. "Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms."

Appendix A

Results from Monte Carlo method

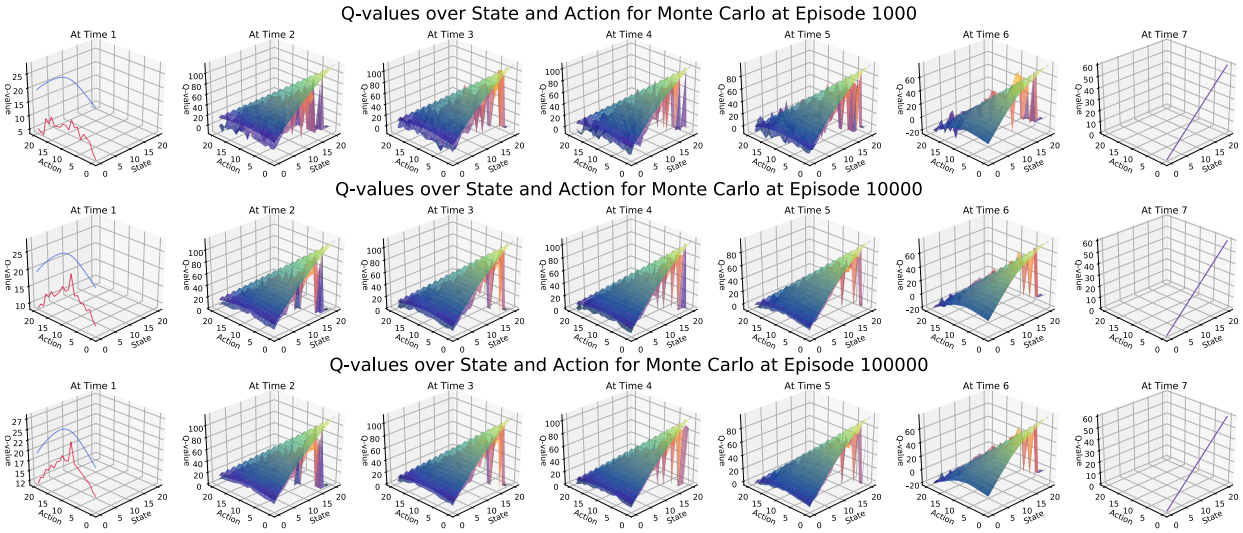

Figure 7: Monte Carlo convergence of the approximate Q-value (red) benchmarked against the optimal Q-value (blue) after 1,000 10,000 and 100,000 episodes

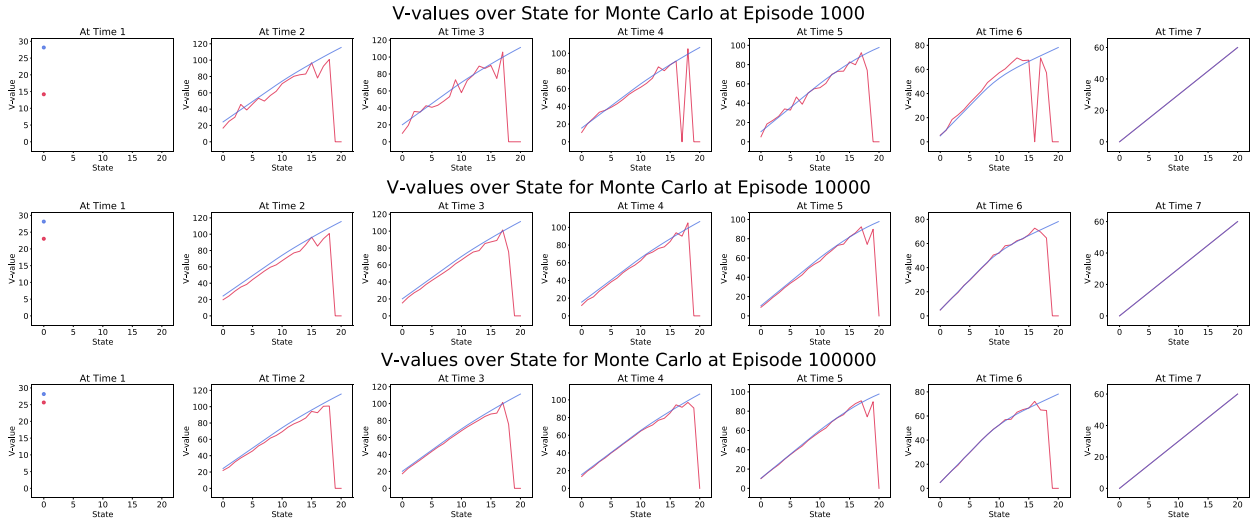

Figure 8: Monte Carlo convergence of the approximate V-value (red) benchmarked against the optimal V-value (blue) after 1,000 10,000 and 100,000 episodes

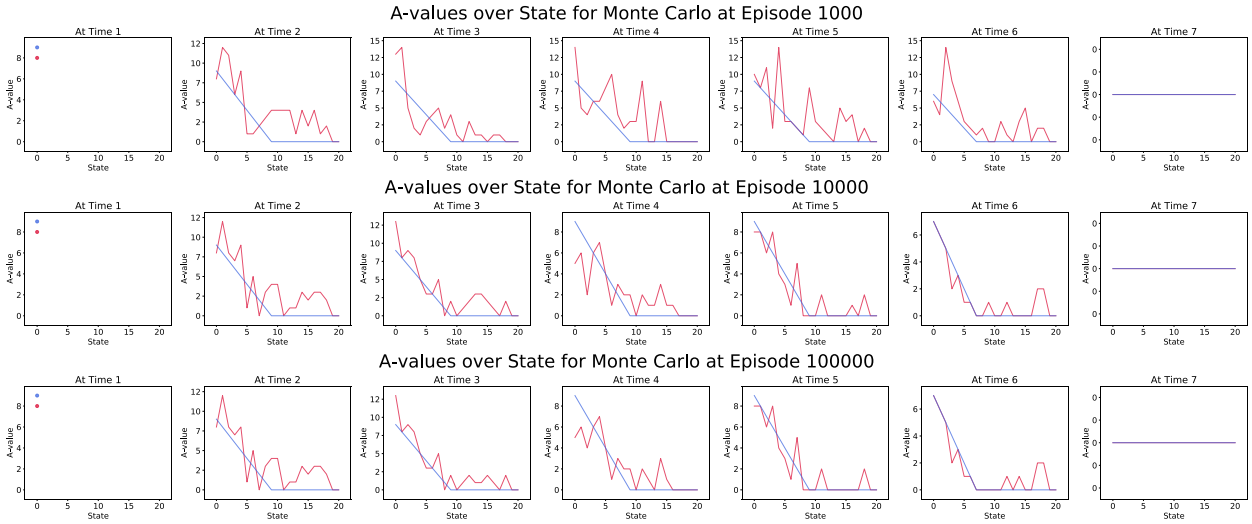

Figure 9: Monte Carlo convergence of the approximate A-value (red) benchmarked against the optimal A-value (blue) after 1,000 10,000 and 100,000 episodes

Results from Sarsa

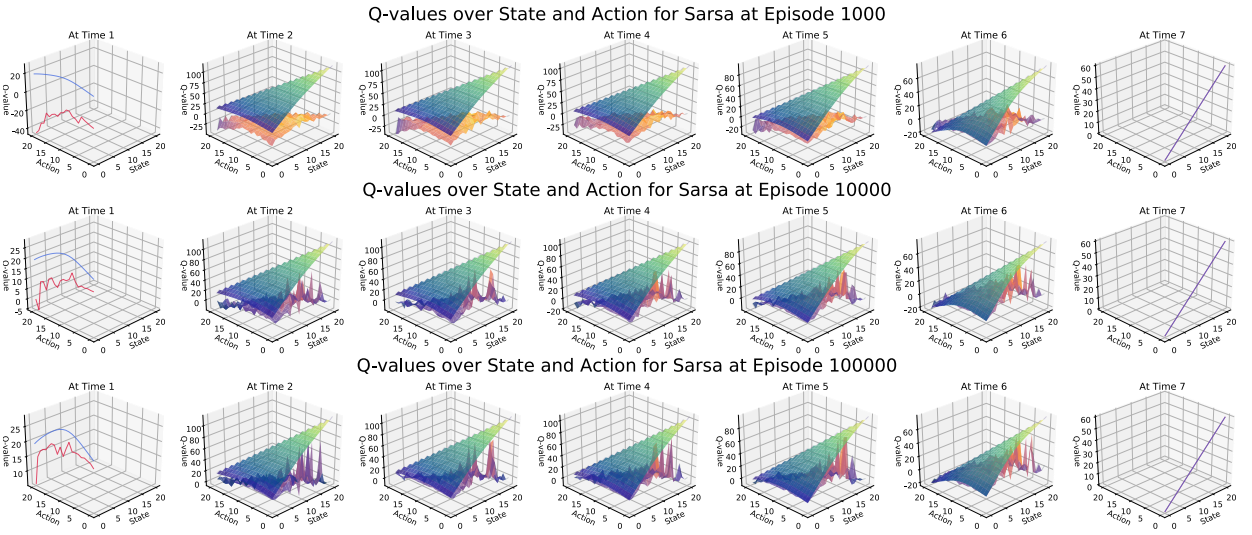

Figure 10: Sarsa convergence of the approximate Q-value (red) benchmarked against the optimal Q-value (blue) after 1,000 10,000 and 100,000 episodes

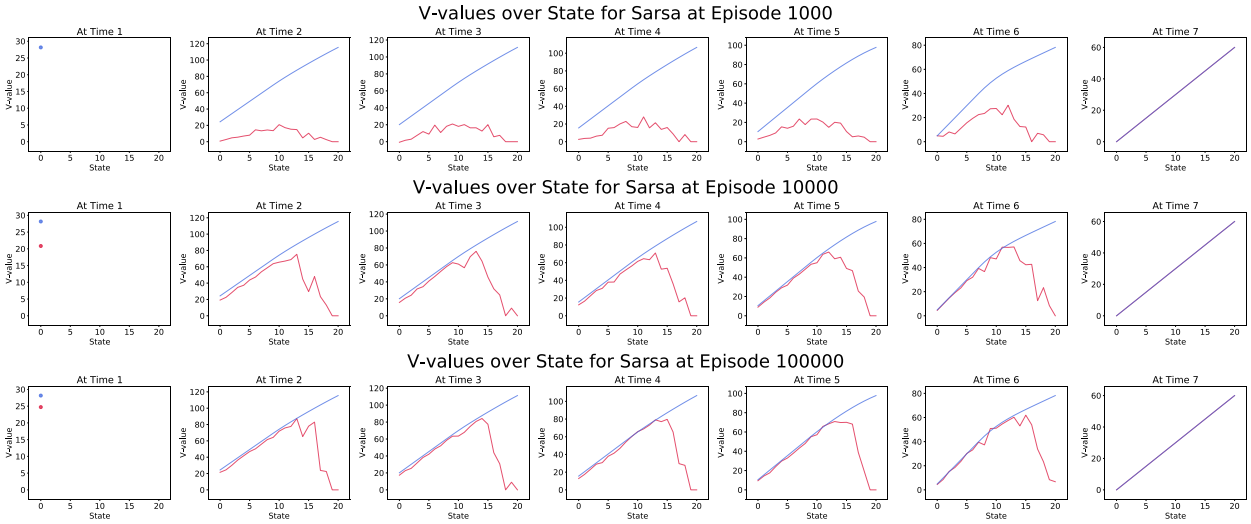

Figure 11: Sarsa convergence of the approximate V-value (red) benchmarked against the optimal V-value (blue) after 1,000 10,000 and 100,000 episodes

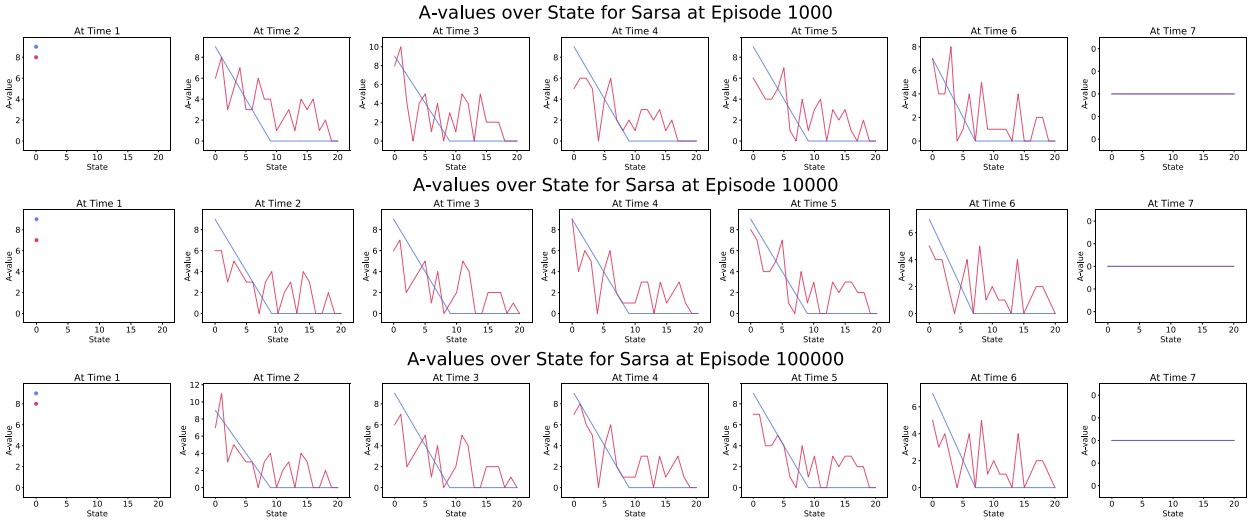

Figure 12: Sarsa convergence of the approximate A-value (red) benchmarked against the optimal A-value (blue) after 1,000 10,000 and 100,000 episodes

Results from Q-learning

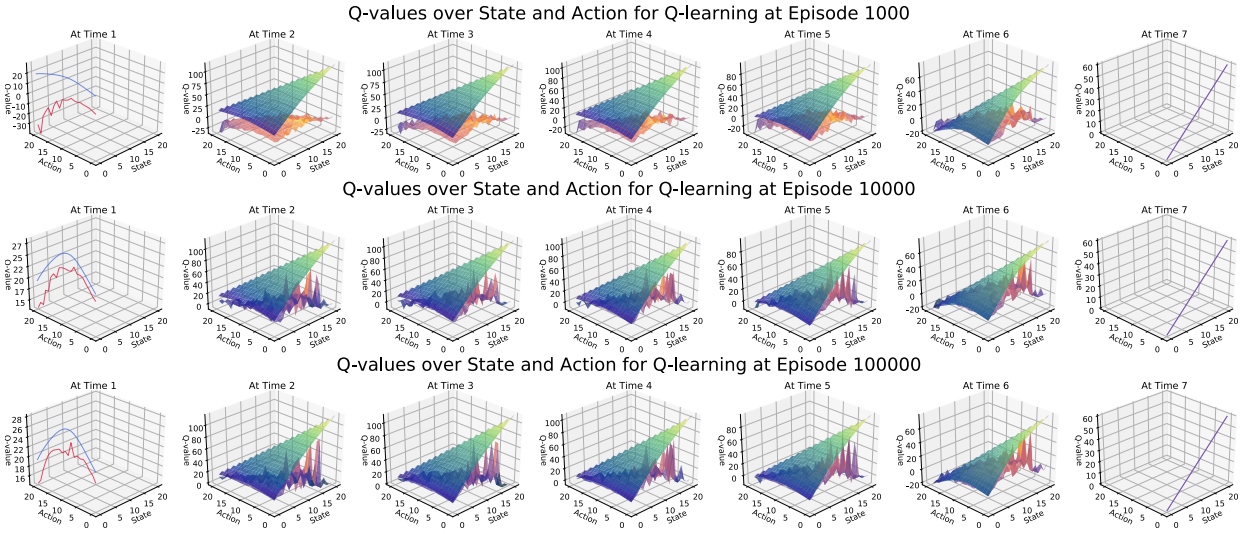

Figure 13: Q-learning convergence of the approximate Q-value (red) benchmarked against the optimal Q-value (blue) after 1,000 10,000 and 100,000 episodes

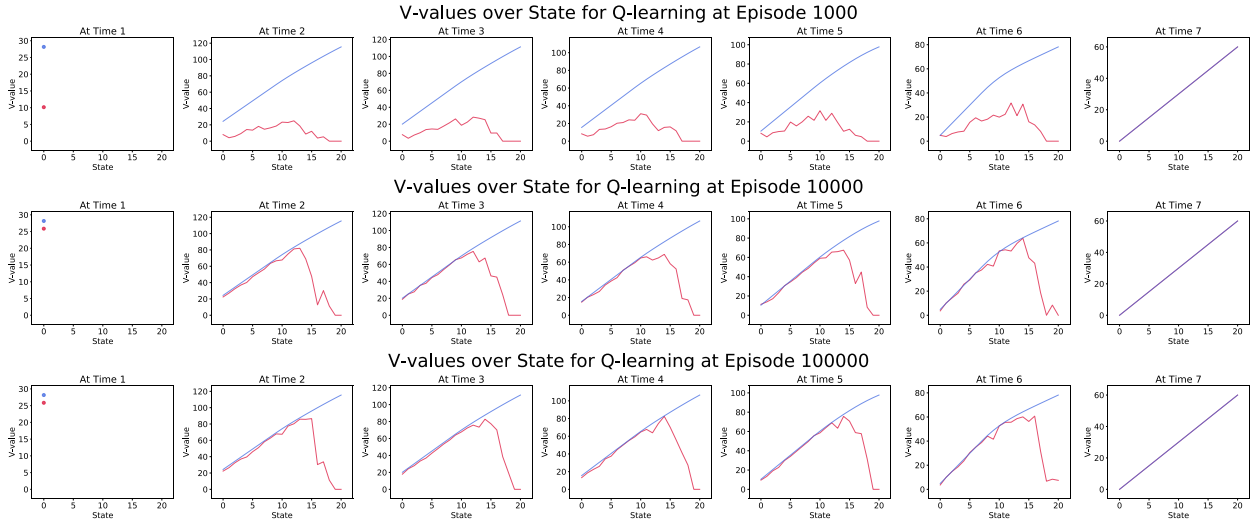

Figure 14: Q-learning convergence of the approximate V-value (red) benchmarked against the optimal V-value (blue) after 1,000 10,000 and 100,000 episodes

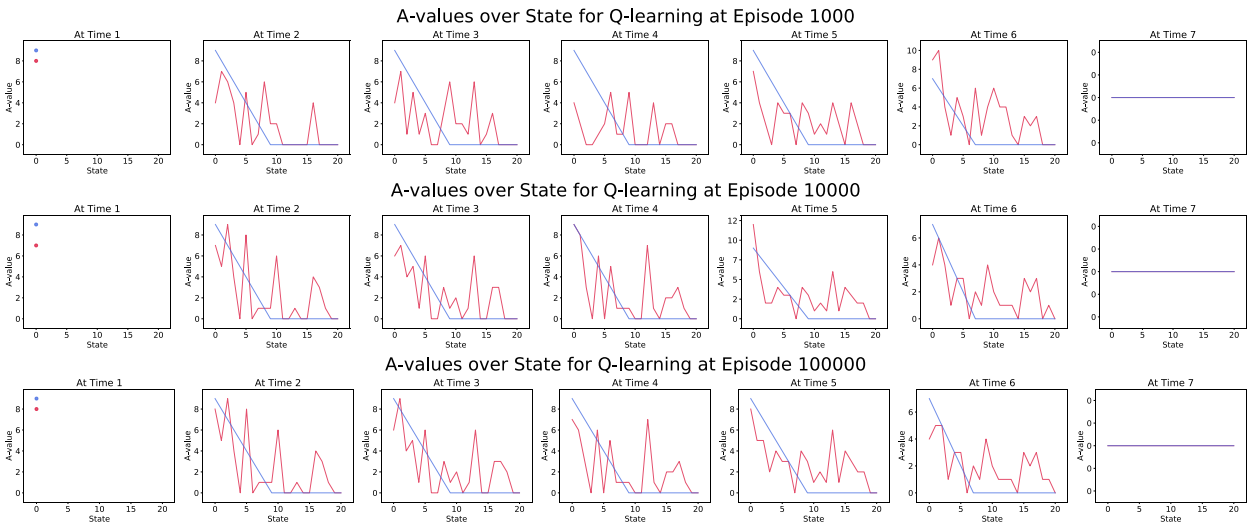

Figure 15: Q-learning convergence of the approximate A-value (red) benchmarked against the optimal A-value (blue) after 1,000 10,000 and 100,000 episodes

Results from Double Q-learning

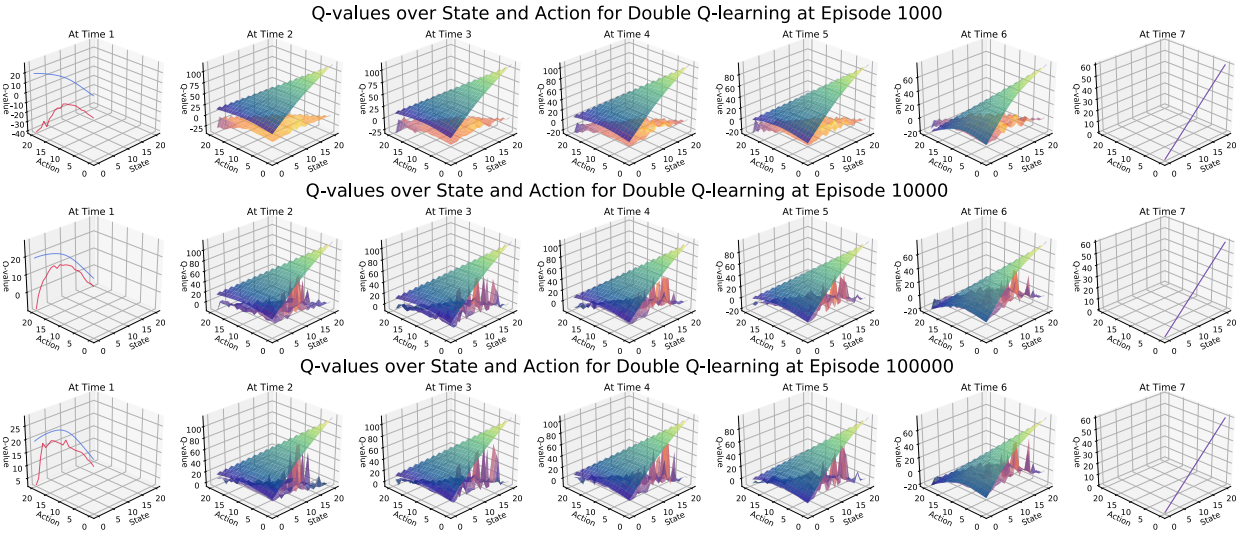

Figure 16: Double Q-learning convergence of the approximate Q-value (red) benchmarked against the optimal Q-value (blue) after 1,000 10,000 and 100,000 episodes

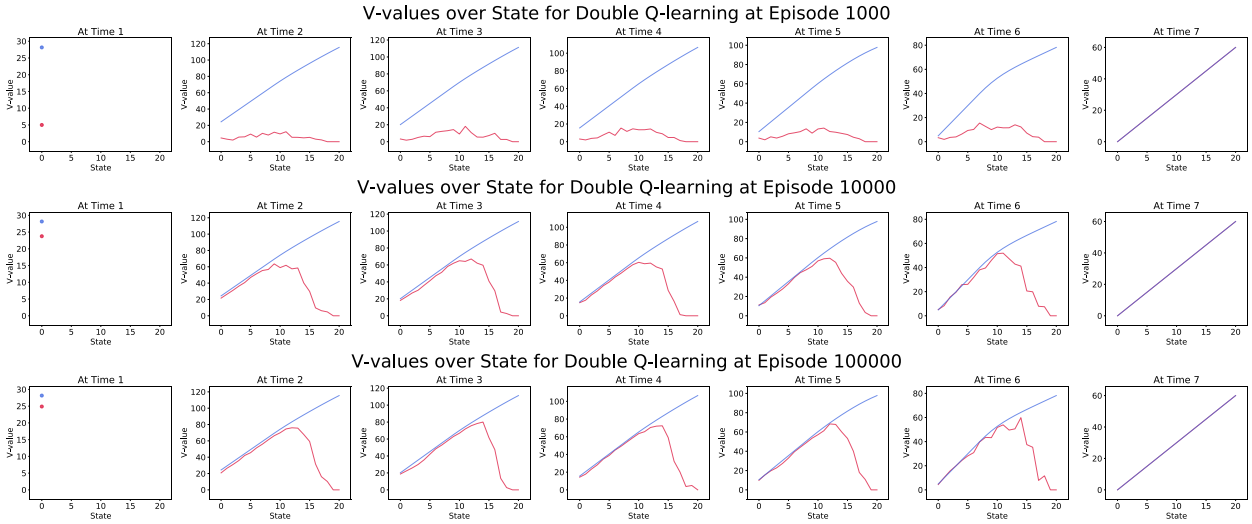

Figure 17: Double Q-learning convergence of the approximate V-value (red) benchmarked against the optimal V-value (blue) after 1,000 10,000 and 100,000 episodes

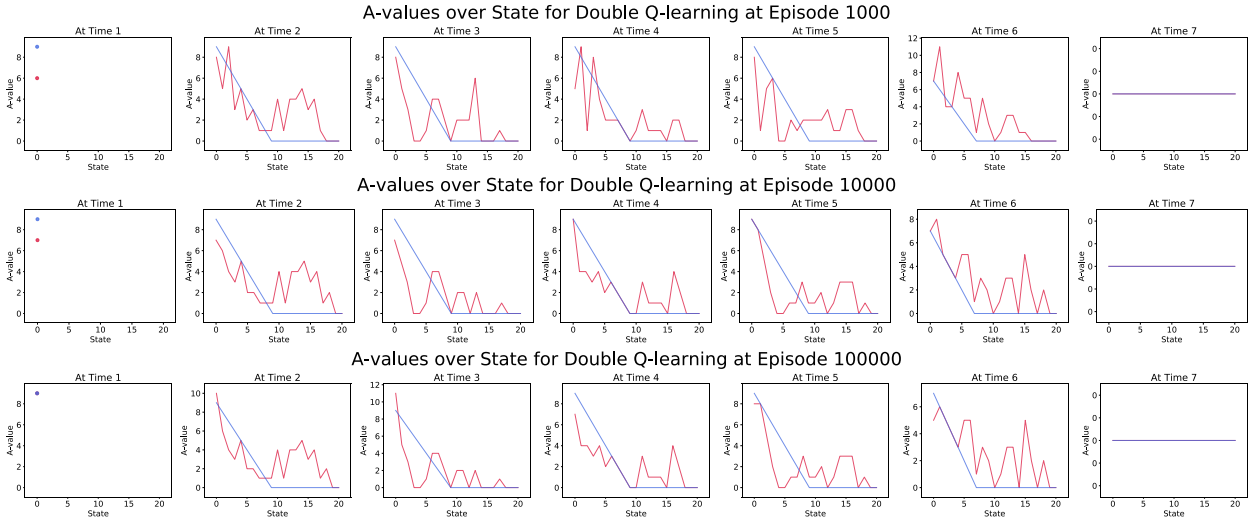

Figure 18: Double Q-learning convergence of the approximate A-value (red) benchmarked against the optimal A-value (blue) after 1,000 10,000 and 100,000 episodes