1. I think the most likely reason is due to the fact that the Monte Carlo method does not have a great advantage in the task environment given in this chapter, especially compared to TD learning. The Monte Carlo method requires an episode to be completed before it starts updating the state, which is very slow to learn, and although the accuracy rate is relatively high, it is really inefficient. In this case, the more efficient TD learning would be more popular.

   I think it is not cost effective to use the Monte Carlo method in a mountain bike task, it will take a lot of time because Monte Carlo will not start learning until he finishes an episode and if he does not happen to gain valid experience then he will continue training. Its performance is completely random.

2. Semi-gradient SARSA

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

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

$\gamma \sum_a \pi(a|s')\hat{q}(s,a,w)$

Semi-gradient Q-learning

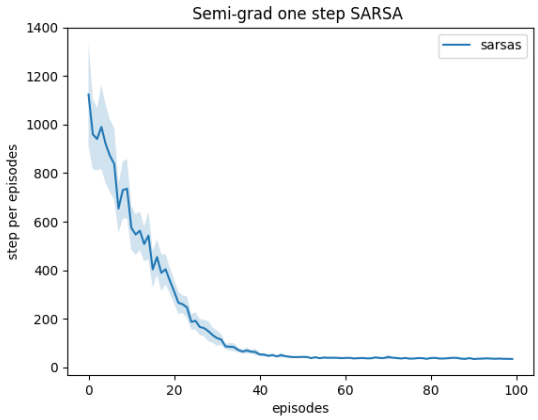
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w}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

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode

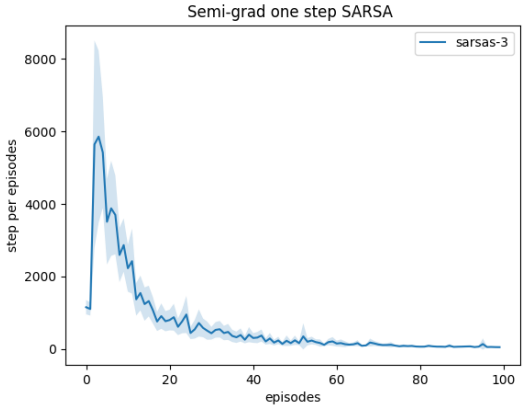
$w \leftarrow w + \alpha[R + \gamma\hat{q}(s', A', w) - \hat{q}(s, A, w)]$.
Choose A' as a function of $q(s', \cdot, w)$
$s \leftarrow s'$
$A \leftarrow A'$.

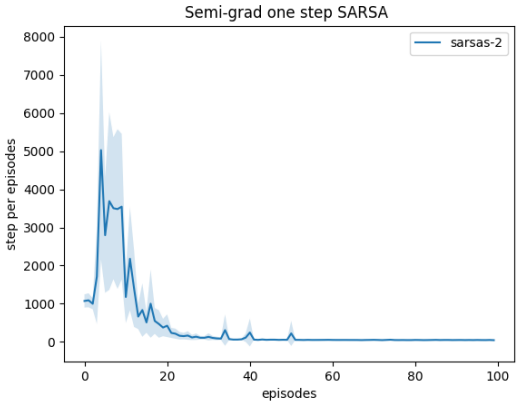
3. (a) I use the one-hot encoding method to express the features, while the linear function estimation is used to predict the Q value, and the gradient is the feature corresponding to the weights.

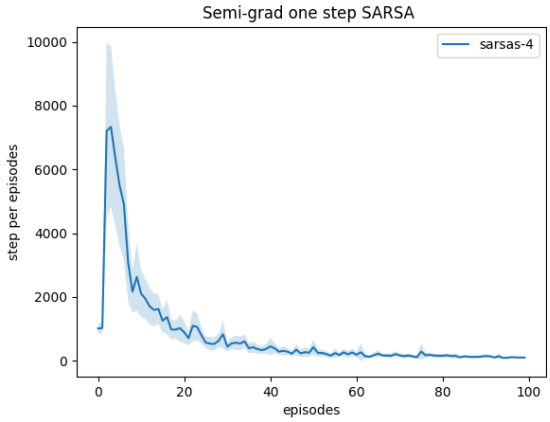
(b)

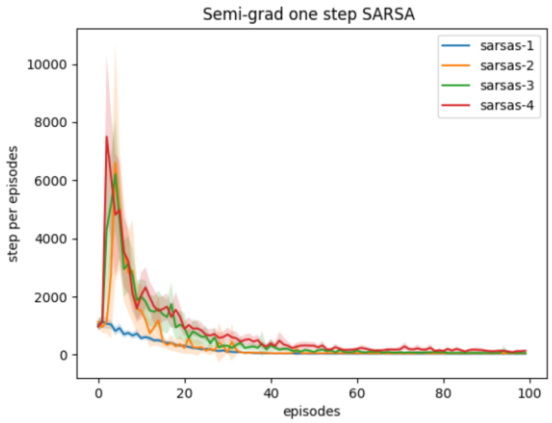


(c) I tried to gather 2, 3 or 4 states into one and see how they behave. Unfortunately, they didn't perform as well and as consistently as I thought (It could also be due to the lack of episodes, maybe I can see a stronger contrast when there are enough episodes), but they did seem to make the function converge a little faster. In comparison, tabular was the best performer instead.



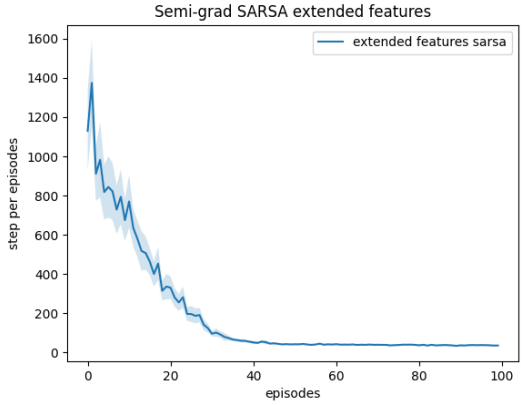




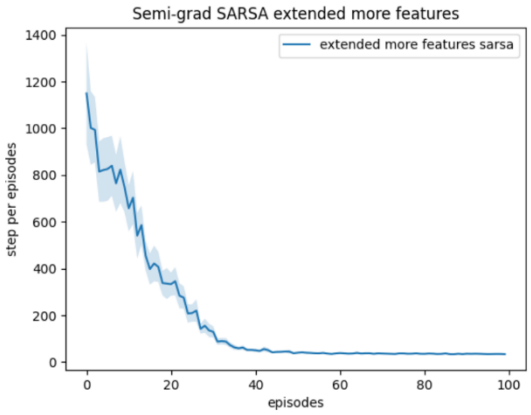


(d) The purpose of the constant term is to reduce the error to make the function as unbiased as possible.

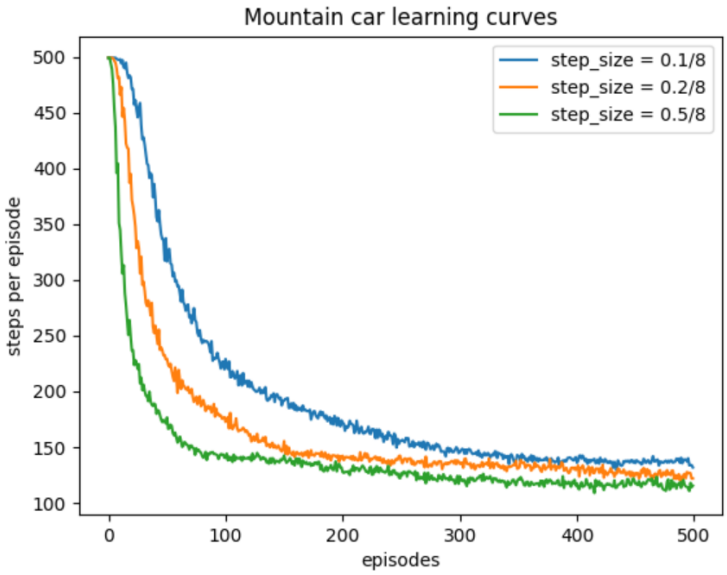
I use vectors to represent the features: (1, the position of the state, the 4 possible actions of the state), which is (1, x, y, 'up', 'down', 'left', 'right'). This method gives about the same result as tabular, which doesn't seem to be a big improvement.



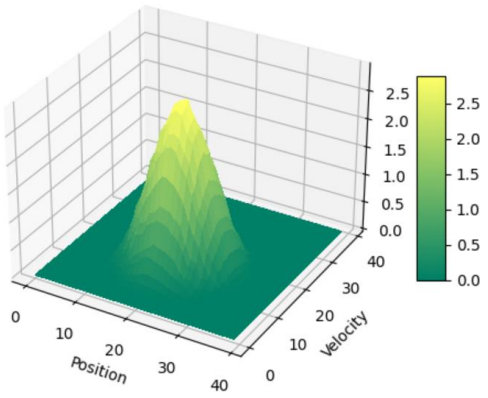
(e) I tried to add 3 more features with information about the Four Rooms domain, namely the distance of the state to the goal position and the start position, and whether the goal state is reached (1 if it is reached), but it does not seem to improve the convergence of the function much, and that's where I'm surprised. I thought the extra features would at least affect the performance a bit. Or maybe I need to add more "useful" features to see the improvement



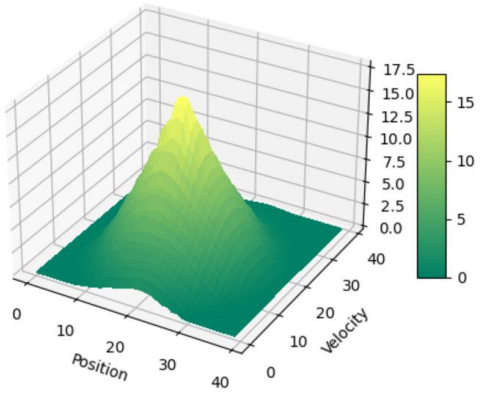
4.



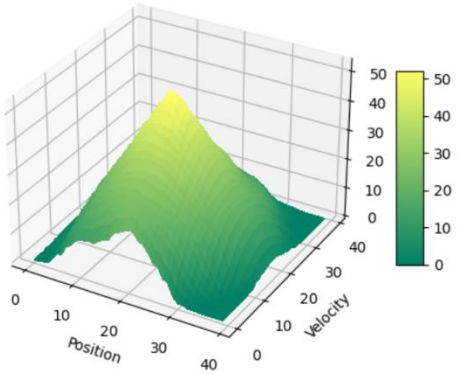
Mountain car episode = 1



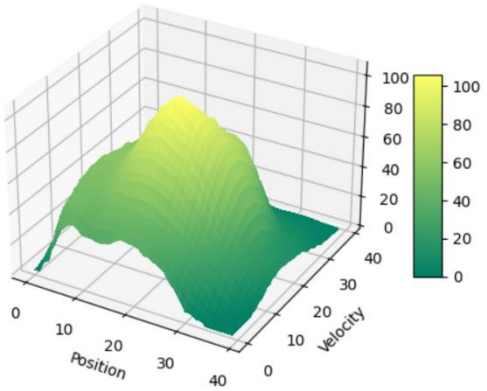
Mountain car episode = 12



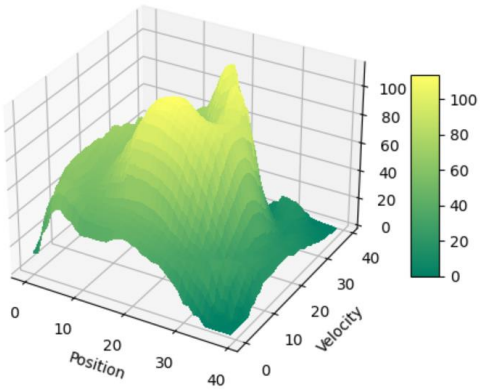
Mountain car episode = 104



Mountain car episode = 1000



Mountain car episode = 9000



Q5. (a) $\overline{VE}(w) \triangleq \sum_{s \in S} \mu(s) [v_\pi(s) - \hat{v}(s,w)]^2$

$\overline{VE}(w) \triangleq \sum_{s \in S} \mu(s) [R + \gamma \hat{v}(s',w) - \hat{v}(s,w)]^2$

$= E[R + \gamma \hat{v}(s',w) - \hat{v}(s,w)]^2$

$w_{t+1} = w_t - \frac{1}{2} \alpha \nabla \overline{VE}(w)$

$\nabla \overline{VE}(w) = 2E[R + \gamma \hat{v}(s',w) - \hat{v}(s,w)] [\nabla \gamma \hat{v}(s',w) - \nabla \hat{v}(s,w)]$

$w_{t+1} = w_t - \alpha E[R + \gamma \hat{v}(s',w) - \hat{v}(s,w)][\nabla \gamma \hat{v}(s',w) - \nabla \hat{v}(s,w)]$

(b) The objective function being optimized is: TD error : $\sum_{s \in S} \mu(s) E[R + \gamma \hat{v}(s',w) - \hat{v}(s,w)]^2$

We want to minimize this mean square TD error, but this is not a good idea because it will cause non-negligible bias that will make the algorithm not converge in the right way and thus give wrong results.

(c) $\overline{BE}(w) \triangleq \sum_{s \in S} \mu(s) [E_\pi[R + \gamma \hat{v}(s',w) | s] - \hat{v}(s,w)]^2$

$= E[E_\pi[R + \gamma \hat{v}(s',w) | s] - \hat{v}(s,w)]^2$

$\nabla \overline{BE}(w) = 2E[E_\pi[R + \gamma \hat{v}(s',w)] - \hat{v}(s,w)][E_\pi(\gamma \nabla \hat{v}(s',w)) - \nabla \hat{v}(s,w)]$

$w_{t+1} = w_t - \alpha E[E_\pi[R + \gamma \hat{v}(s',w)] - \hat{v}(s,w)][E_\pi(\gamma \nabla \hat{v}(s',w)) - \nabla \hat{v}(s,w)]$

(d) The problem is that in this learning rule, expectations are multiplied, but "The expectation of a product is not necessarily equal to the product of the expectations." To make the result unbiased, we would need two independent samples of the next state, but the sampling of both expectations depends on the calculation of the next state, s'. We cannot sample both at the same time, therefore, we cannot make this algorithm unbiased.

However, in some special cases, we may be able to overcome this problem, or in other words, the problem does not exist in that case. For example, in the case of deterministic environmental conditions, the expectations are multiplicable, since their transition to the next state is the same and therefore the samples are identical. There would not be the problem we had before. Or we could 'simulate' two independent samples of the next state separately, but this is something that cannot happen in a real environment.