EX-07

1. **1 point.** (RL2e 10.1) *On-policy Monte-Carlo control with approximation.*
   Written: We have not explicitly considered or given pseudocode for any Monte Carlo methods in this chapter.
   What would they be like? Why is it reasonable not to give pseudocode for them?
   How would they perform on the Mountain Car task?

The Monte Carlo approach bears resemblance to the on-policy Monte-Carlo control method, as well as to the Temporal Difference (TD) learning method, which is indicated by using n=T. There is little variation in the pseudocode between these methods, thus providing pseudocode is not deemed necessary.

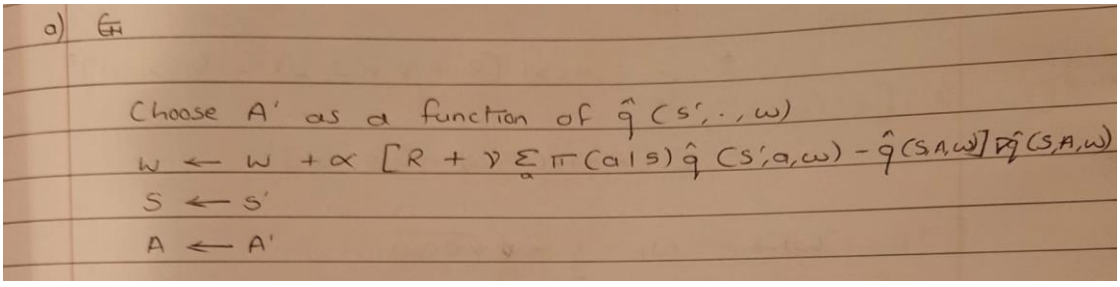
The Monte Carlo approach applied to the Mountain Car problem may incur higher costs, as the learning process only kicks off after finishing an entire episode. Additionally, every step that fails to reach the goal state incurs a negative reward, elevating the overall cost and extending the time needed to achieve convergence when compared to learning methods like TD or SARSA.

2. **1 point.** (RL2e 10.2) *Semi-gradient expected SARSA and Q-learning.*
   Written:
   (a) Give pseudocode for semi-gradient one-step Expected Sarsa for control.
   (b) What changes to your pseudocode are necessary to derive semi-gradient Q-learning?

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

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

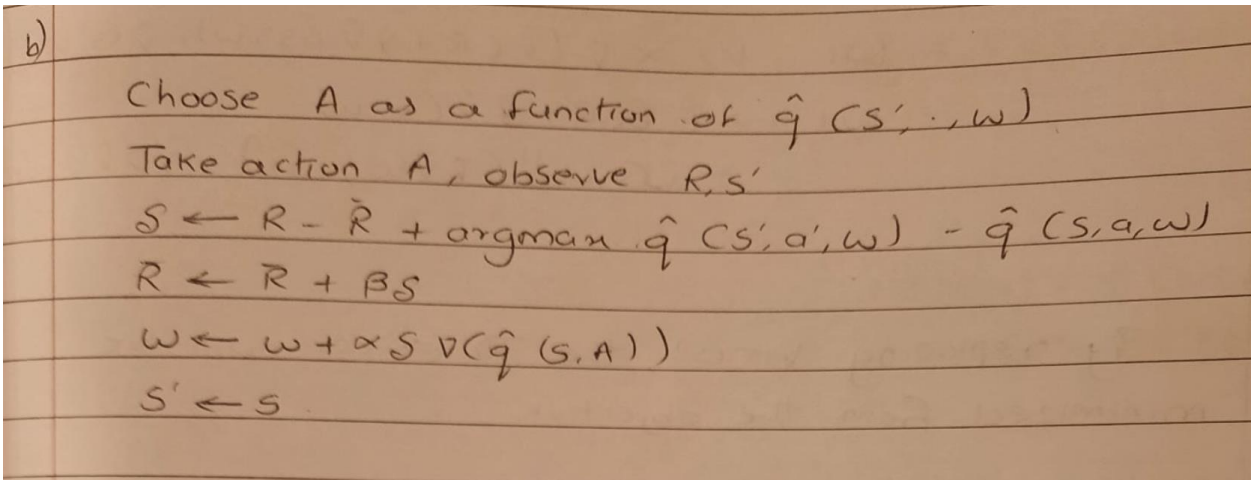
Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$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



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 $\alpha, \beta > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
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., ε-greedy)
- $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
- $\bar{R} \leftarrow \bar{R} + \beta\delta$
- $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
- $S \leftarrow S'$
- $A \leftarrow A'$

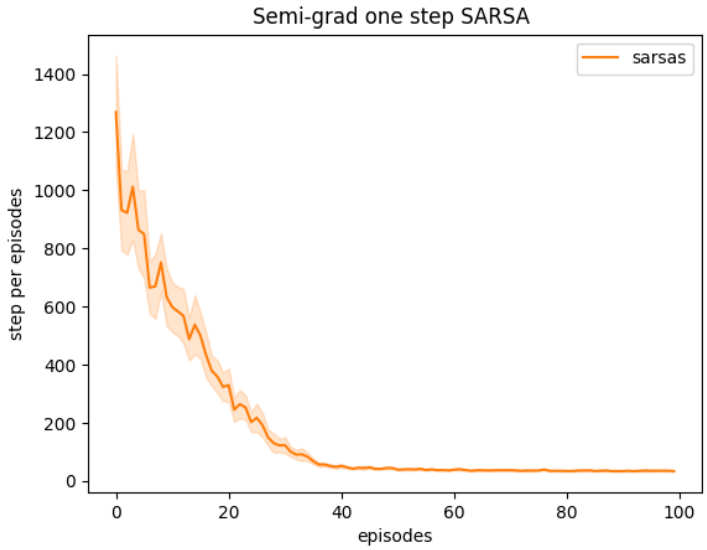


3. **4 points.** *Four rooms, yet again.*
Let us once again re-visit our favorite domain, Four Rooms, as implemented in Ex4 Q4.
We will first explore function approximation using *state aggregation*.
Since this domain has four discrete actions with significantly different effects, we will only aggregate states; different actions will likely have different Q-values in the same state (or set of states).

a)

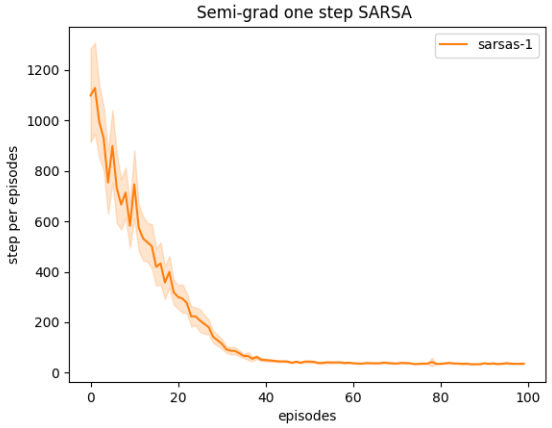
I design to use different weight vectors for each aggregate states. The weight vector has the same length with aggregations. By using one-hot coding method, the gradient of weight is correlated to the specific feature that needs to be used.

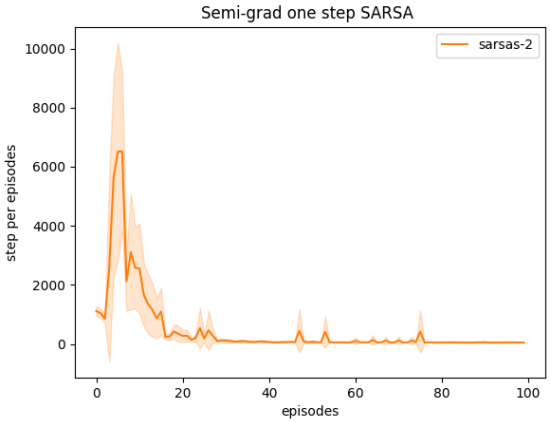
b) Code/plot

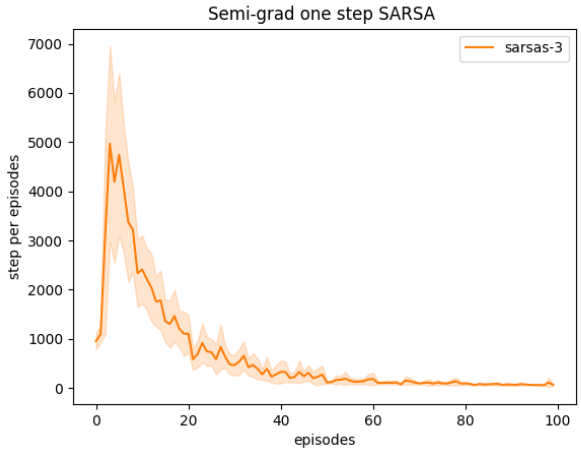


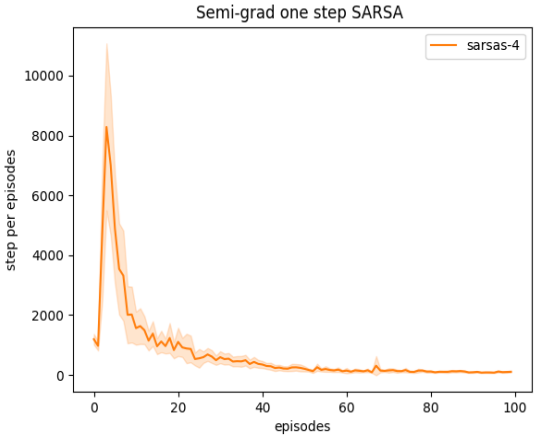
c)

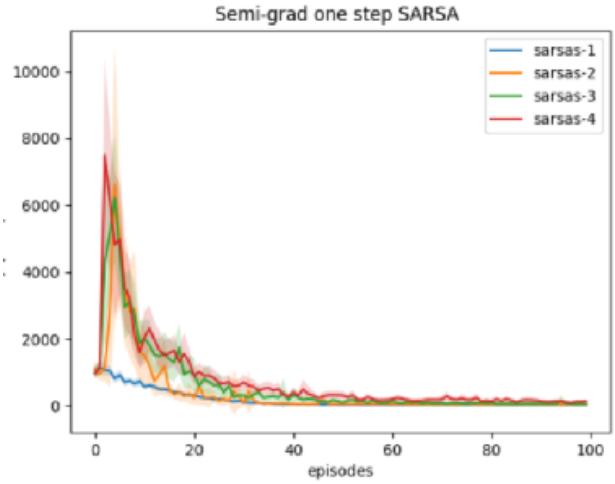
I experimented with combining 2, 3, or 4 states to observe the resulting dynamics. Regrettably, the performance and consistency did not meet my expectations, which may be attributed to an insufficient number of episodes. It's possible that a clearer difference might emerge given a larger episode count. However, it did seem to expedite the convergence process marginally. In contrast, using a tabular approach turned out to be the most effective in terms of performance.


Semi-grad one step SARSA
sarsas-1
step per episodes
episodes


Semi-grad one step SARSA
sarsas-2
step per episodes
episodes

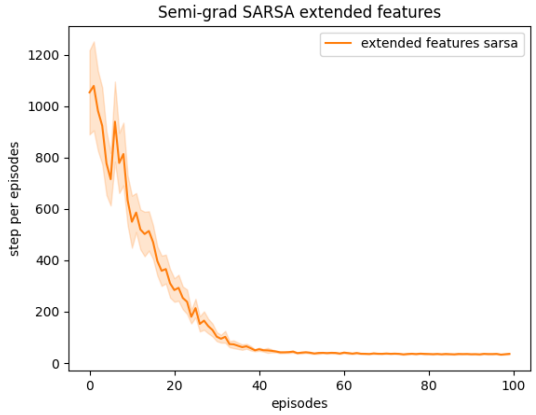

Semi-grad one step SARSA
sarsas-3
step per episodes
episodes


Semi-grad one step SARSA
sarsas-4
step per episodes
episodes


Semi-grad one step SARSA
sarsas-1
sarsas-2
sarsas-3
sarsas-4
episodes

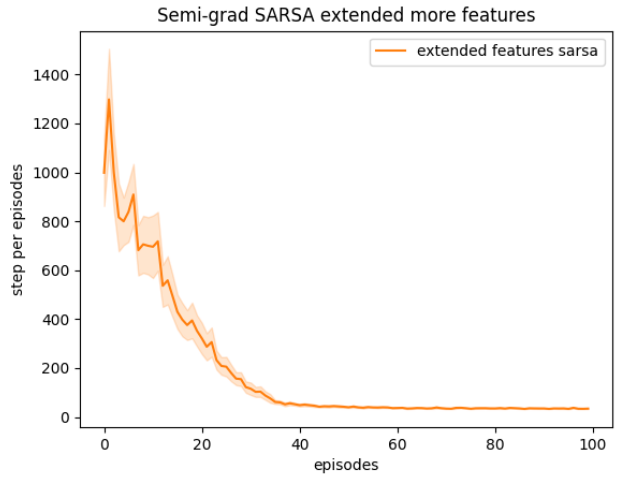
d)The constant term's role is to minimize error and foster a function that is as unbiased as possible.

The features are encoded using vectors that include a constant term, the state's coordinates, and the four potential actions within that state—represented as (1, x, y, 'up', 'down', 'left', 'right'). Utilizing this vector representation yields results similar to the tabular approach, indicating no significant enhancement in performance.



e)

In an attempt to enrich the feature set for the Four Rooms domain, I incorporated three additional features, including the state's distance from both the goal and the start positions, and a binary indicator signifying whether the goal has been attained (1 indicating success). Contrary to my expectations, these new features didn't significantly enhance the function's convergence rate, which was quite unexpected. I had assumed that the inclusion of extra features would, at the very least, slightly influence performance. It appears I may need to integrate more 'useful' features to observe any notable improvement.



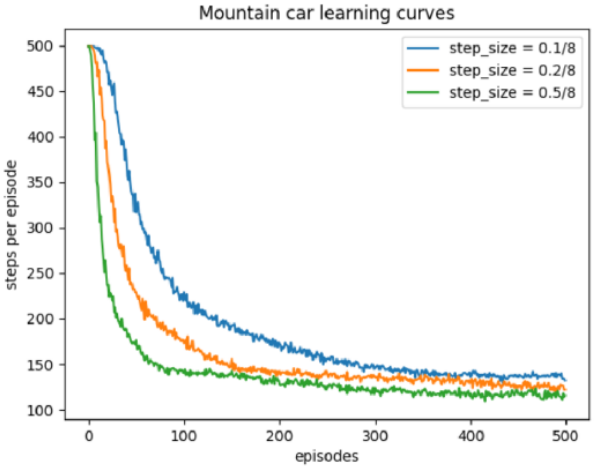
4. **[CS 5180 only.] 2 points.** *Mountain car.*

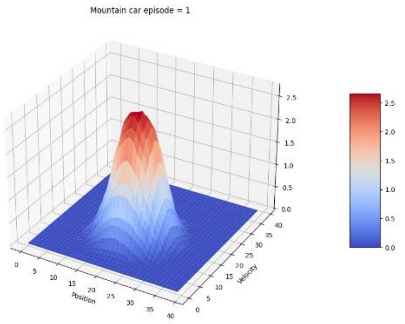
   (a) **Code/plot:** Read and understand Example 10.1.
   Implement semi-gradient one-step SARSA for Mountain car, using linear function approximation with the tile-coding features described in the text. Reproduce Figures 10.1 and 10.2.
   Instead of writing your own environment and features, we recommend that you use the implementation of Mountain Car provided by Gymnasium, and refer to the footnote on p. 246 for an implementation of tile coding. Make sure you use the discrete-action version (`MountainCar-v0`):
   `https://gymnasium.farama.org/environments/classic_control/mountain_car`

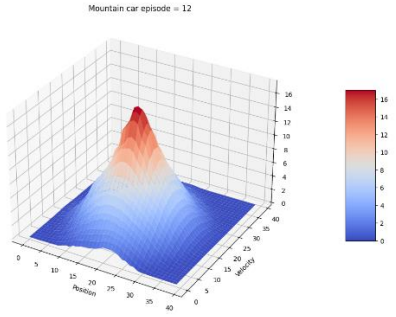
Some notes on Mountain Car and reproducing figures:

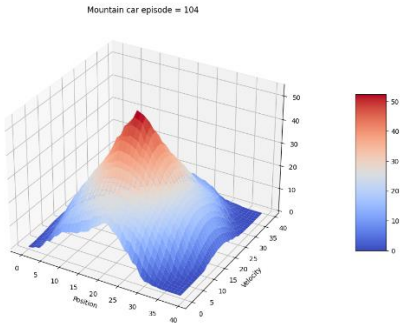
- The implementation in Gymnasium is close to the book's description, but it has a timeout of 200 steps per episode, so the results you get may be different from that shown in Figures 10.1 and 10.2. This is fine and expected. (If you see footnote 2 on p. 246, you will also see that Figure 10.1 was generated using semi-gradient SARSA(λ) instead of semi-gradient SARSA.)
- For visualizing the cost-to-go function (Figure 10.1), you can use plotting tools like `imshow` instead of showing a 3-D surface, if you wish.
- Since episodes time out after 200 steps, for the first subplot of Figure 10.1, just visualize the cost-to-go function at the end of the first episode, instead of after step 428 as shown.
- If your implementation is too slow, you can omit visualizing the cost-to-go function at episode 9000 (the final subplot in Figure 10.1).
- When replicating Figure 10.2, it is fine if your vertical axis is a regular linear scale (vs. log scale as shown), since the maximum will be 200 steps per episode. Also, if your implementation is too slow, you can do fewer than 100 trials (e.g., 10 is okay).

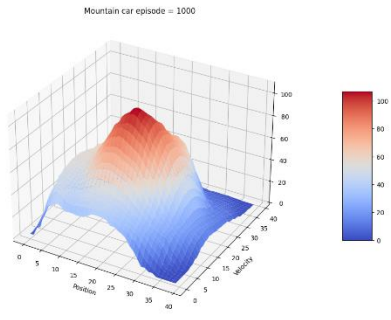
The following questions present many opportunities for extra credit. Feel free to do any of these, at any time. You may submit these late without penalty, until the end of the month (3/31). If you do submit these later than your main assignment, please inform the course staff via Piazza.

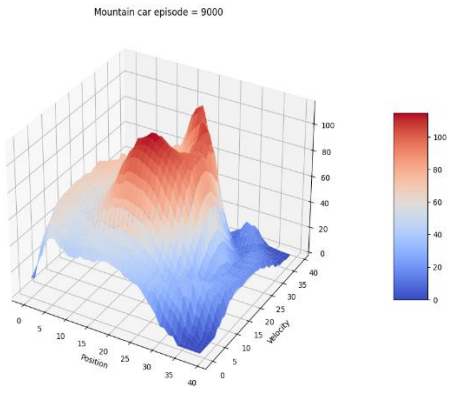



Mountain car episode = 1
Position
Velocity


Mountain car episode = 12
Position
Velocity


Mountain car episode = 104
Position
Velocity


Mountain car episode = 1000
Position
Velocity


Mountain car episode = 9000
Position
Velocity

5.

5)

a) $V_\pi(s) = R + \gamma \hat{v}(s', w)$

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

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

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

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

$$= w_t - 1/2 \, \alpha \nabla [\mathbb{E}(R + \gamma \hat{v}(s', w) - \hat{v}(s, w))]^2$$

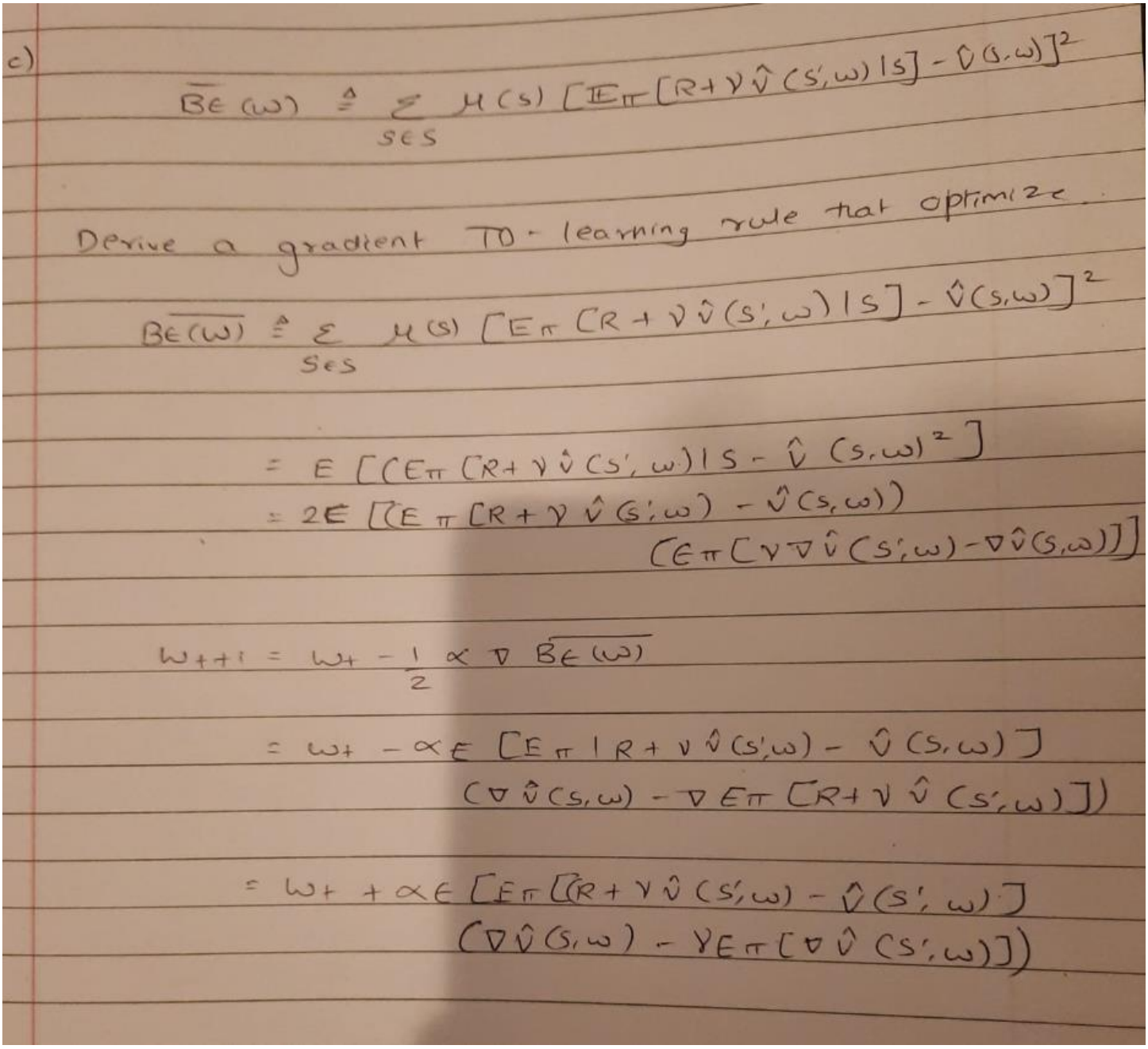
$$= w_t + \alpha \mathbb{E}[(R + \gamma \hat{v}(s', w) - \hat{v}(s, w))(\nabla \hat{v}(s', w_t) - \gamma \nabla \hat{v}(s', w_t))]$$

b)

By replacing $V_\pi(s)$, the TD error will be minimized from the objective.

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

Not a good idea to involve expectation over s' as loss of fun^n.



d) In the method under consideration, the challenge lies in generating two expectations. To acquire an unbiased sample, independent samples from the subsequent state are necessary. Yet, it's not guaranteed that two samples are identical unless they're mutually independent. In real-world scenarios involving typical interactions, only one expectation is necessary. Hence, this algorithm may exhibit bias and converge to an incorrect value. In deterministic environments, where the next state transition is fixed, the two samples would be identical, allowing the algorithm to function correctly. Conversely, in simulated environments where independent samples can be used for different expectations, the algorithms have the potential to converge to the minimum value.