Imperial College London

COURSEWORK 1

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Reinforcement Learning

Kristian Apostolov

02547890

November 3, 2023

1 Dynamic Programming

1.1 Results

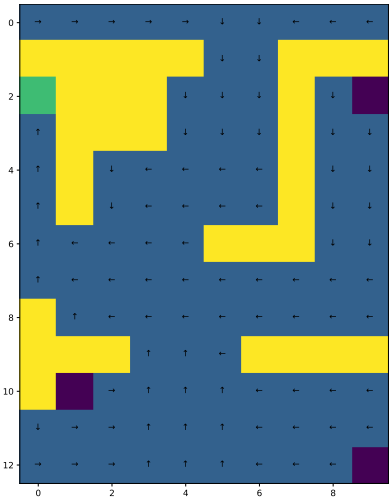


Figure 1: Dynamic Programming *(Policy)*

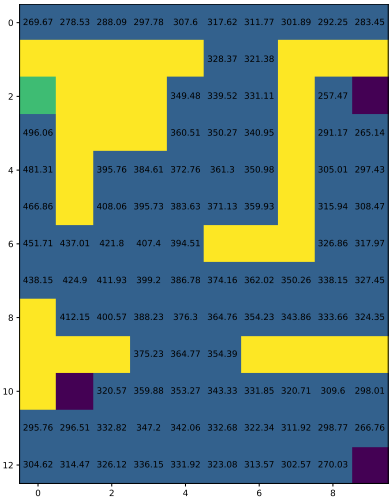


Figure 2: Dynamic Programming *(Values)*

1.2 Algorithm and Parameter Selection

The submission implements both Policy Iteration (PI) and Value Iteration (VI). Both methods produce the same optimal policy and values, however PI needs much more iterations to converge compared to VI as it needs to evaluate a policy on each value update (Fig. 3). In contrast VI selects the action which yields the highest returns (it assumes a greedy policy) in order to update the values.

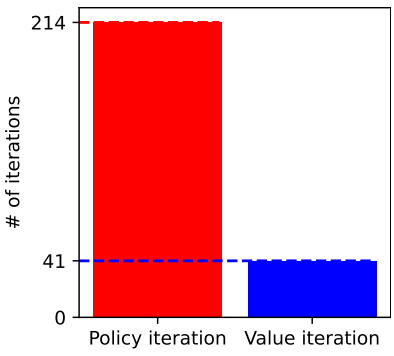


Figure 3: Number of iterations of PI and VI

The Value Iteration (VI) algorithm employs a single hyperparameter: **threshold**. Initially, the chosen metric for fine-tuning this threshold was the count of iterations, as depicted in Fig. 4. The underlying assumption was that, beyond a minimal threshold, the algorithm would consistently converge in the same number of iterations. This proved ineffective because of the continuous nature of the error between new and old values. This difference does not ever become exactly zero; rather, it tends towards zero until limited by machine precision. Narrowing down the precision simply increases the number of iterations without providing clarity on its actual benefit.

A more insightful measure is **the number of policy updates**. The algorithm uses a greedy policy which is essentially reduced to a 1-hot encoding where the optimal action is denoted by a value of 1, while all other actions get a 0. This means that if there are no further policy updates, the optimal policy has been successfully determined, regardless of the precision of the state-action values. In Fig. 4 the policy updates appear to stabilize around a threshold of 10, but setting it at 1 guarantees its effectiveness across varied environments, demanding only a negligible increase in iterations.

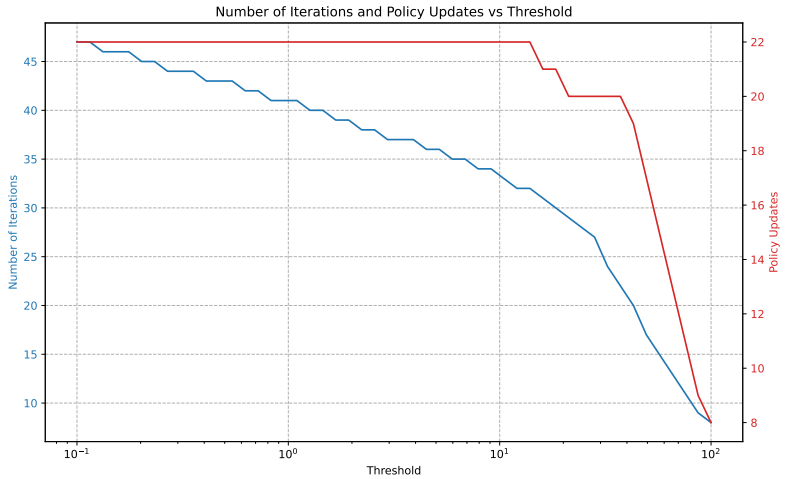


Figure 4: Threshold Tuning

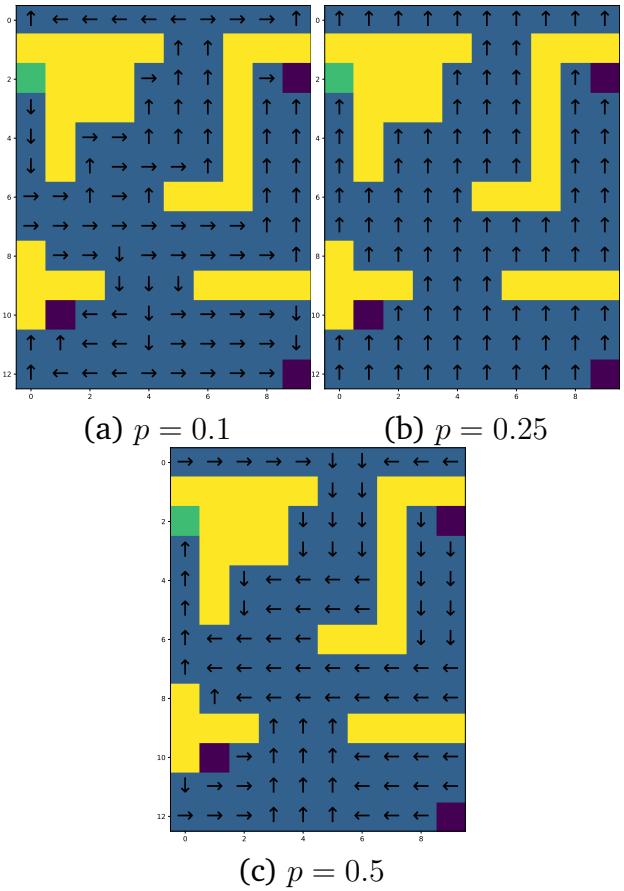
1.3 Influence of Environment Parameters

The effect of the probability of success p is more prominent on the resulting policy as shown on Fig. 5.

(a) **The policy is reversed.** Due to the low probability of an action succeeding, the best course of action is to go towards the state with the lowest value, as it will minimize the risk of landing on it.

(b) **The policy is random.** This is because the values for all 4 actions in any given state are the same: the average of the values of neighbouring states. Since all action values are equal, np.argmax arbitrarily selects the first action, resulting in the appearance of a preference for 'up' movements.

(c) **The policy is normal**. For any value of p that is greater than 0.25, choosing the action towards the highest-value is the optimal decision.



(a) $p = 0.1$ (b) $p = 0.25$

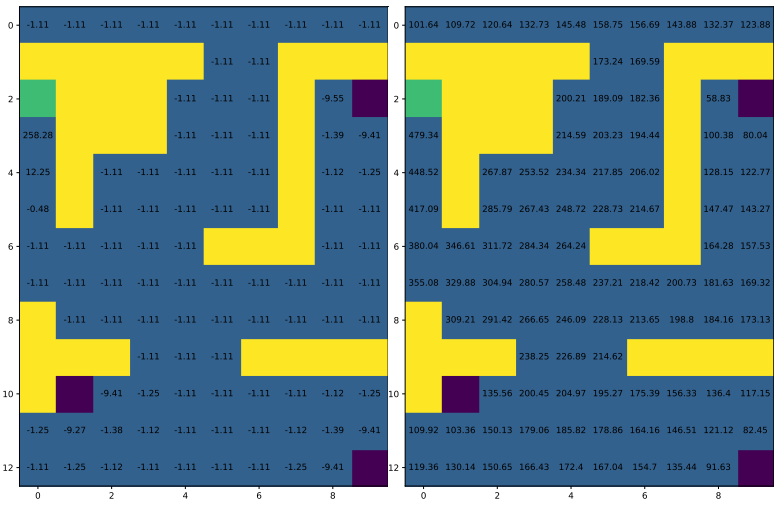
(c) $p = 0.5$

Figure 5: Effect on results with different p

On the other hand, the effect of γ is better depicted using the actual values (See Fig. 6).

(a) The impact of a reward on a state's value diminishes with increasing distance from the target. Nearby states experience a stronger influence due to closer proximity, which decays exponentially as the distance grows. The value of any given state is thus predominantly influenced by its immediate neighbors, with the reward's effect on more distant states being considerably weaker.

(b) Conversely, having a high γ allows for values to affect states further away which is the desired effect.



(a) $\gamma = 0.1$ (b) $\gamma = 0.98$

Figure 6: Effect on results with different γ

Distance *is used to refer to spatial distance, but can be extended to more general definition outside the maze context as a* ***metric of the dissimilarity between two states.***

2 Monte-Carlo

2.1 Results

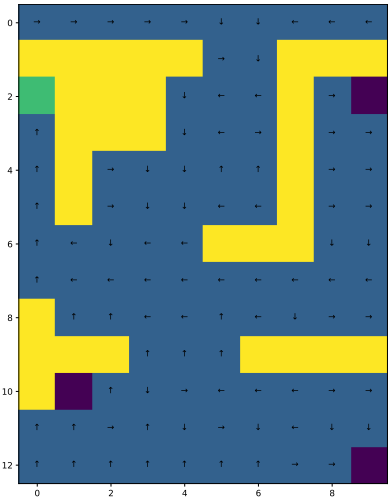


Figure 7: Dynamic Programming *(Policy)*

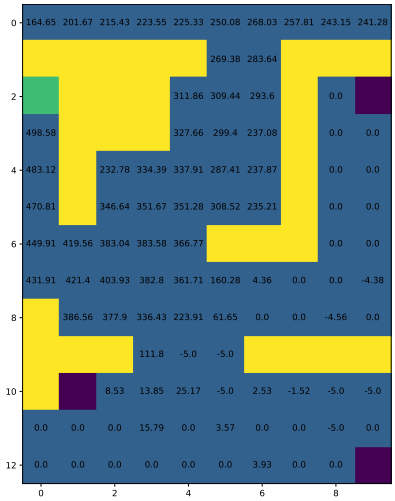


Figure 8: Dynamic Programming *(Values)*

2.2 Algorithm and Parameter Selection

The submission implements the First-Occurrence Iterative Monte-Carlo (IMC) algorithm. This approach was chosen over Batch Monte-Carlo (BMC) primarily to avoid the need for tuning an additional hyperparameter, batch_size. While IMC offers a more straightforward implementation, it might not harness the computational advantages of array vectorization that

BMC can, potentially making BMC more suitable for more complex environments. The main assumption made for this algorithm is that given the discrete and relatively small state and action space the speed drawback would be negligible. Additionally, for this specific task First-Occurrence was chosen because the return of the first occurrence already incorporates the rewards of following occurrences (discounted by a power of γ), which *should be* sufficient.

Two hyperparameters were tuned: α - which determines how much of the error should be incorporated when updating the Q-values. ϵ - which determines the extent of **exploration**. The factor of allowed exploration determines how often the agent will take a sub-optimal action according to the current policy in order to explore the environment more extensively and reduce the risk of prematurely converging to a sub-optimal policy. Figure 9 shows the Mean-Squared Error (MSE) of Q over different values of α and γ. The final values chosen for the algorithm are $\alpha = 0.1$ and $\epsilon = 0.1$ because they yield fastest and most stable convergence. The impact of the hyperparameter values is looked into more detail in Section 3.3.

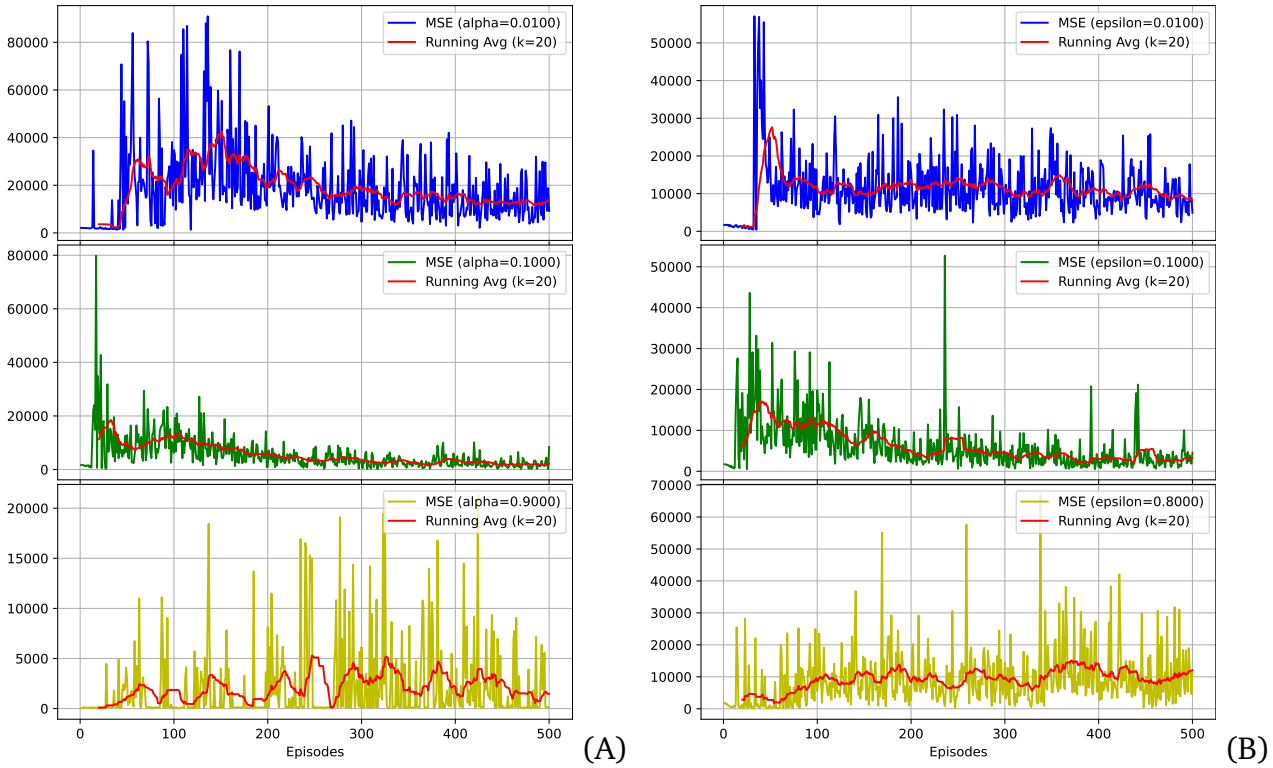


Figure 9: Fine-tuning MC hyperparameters

2.3 Learning Curve

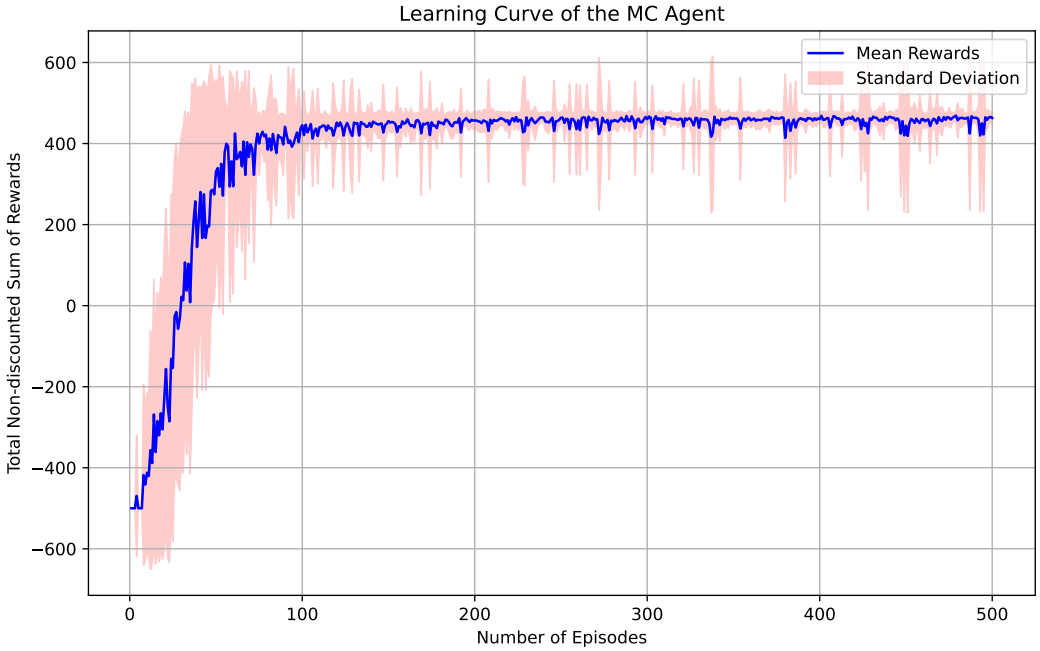


Figure 10: Learning Curve of On-line MC over 25 runs

3 Temporal Difference

3.1 Results

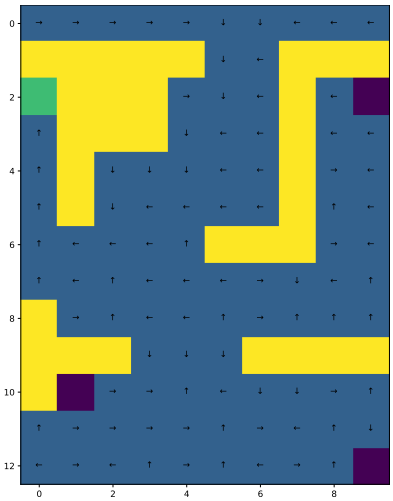


Figure 11: Temporal Difference *(Policy)*

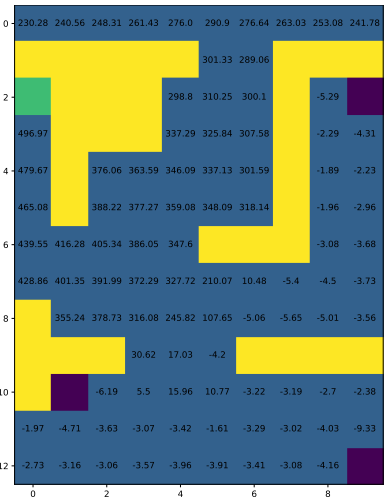


Figure 12: Temporal Difference *(Values)*

3.2 Algorithm and Parameter Selection

The submission implements both *SARSA* and *Q-learning*. Figure 13 compares the rewards produced by each algorithm. The comparison shows that both SARSA and Q-Learning converge at the same rate, with reward functions behaving very similarly, indicating no significant advantage for either algorithm in terms of convergence speed and reward behavior. However, SARSA generally has an advantage over Q-Learning in this specific stochastic environment due to its on-policy nature, allowing it to better account for the possibility of actions failing. In contrast, Q-Learning adopts a more 'optimistic' and greedy approach, selecting

the action with the highest Q value. Despite the similar convergence patterns, the SARSA algorithm has been chosen to produce the final report, likely due to its ability to learn more robust policies in this context.

The algorithm employs two hyperparameters $\alpha = 0.1$ and $\epsilon = 0.1$. The following section demonstrates the reasoning behind these numbers.

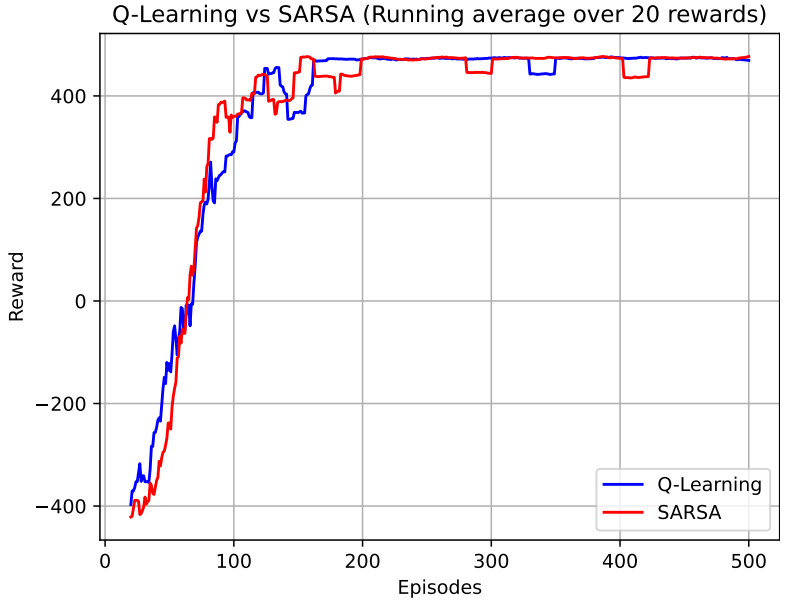


Figure 13: Q-learning vs SARSA rewards

3.3 Impact of α and ϵ

In figure 15 the impact of α and ϵ over the MSE of SARSA can be observed. The episode-wise MSE for episode T is defined in the following way:

$$\text{MSE} = \frac{1}{|T|}\sum_{t=1}^{|T|}(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))^2 \tag{1}$$

Figure 14: Episode Mean-Squared Error

This error is generally more volatile in TD learning compared to MC methods because the error in TD is influenced by the next immediate state and reward, which can vary significantly from one step to the next. In contrast, the error in MC is calculated as the difference between the estimated value and the actual sum of rewards obtained from the entire episode, which tends to average out the variations over all the states visited in the episode. This is why for demonstration purposes the error was tracked over 5000 episodes to give more time for the algorithm to converge and allow for trends in the plots to become more apparent.

α values:

1. $\alpha = 0.01$: The Mean Squared Error (MSE) starts at a relatively high value and decreases slowly over episodes. It demonstrates a gradual reduction in error but takes a considerable amount of time (many episodes) to converge to a stable value. This is indicative of the effect of a low learning rate, where the agent learns very slowly, making small adjustments to its Q-values at each step.

2. $\alpha = 0.10$: The MSE starts at a high value but then rapidly drops to a much lower and

stable value. The speed of convergence is much faster compared to the low α graph. This represents an optimal learning rate, where the balance between exploration of new knowledge and exploitation of existing knowledge is maintained, allowing for efficient learning.

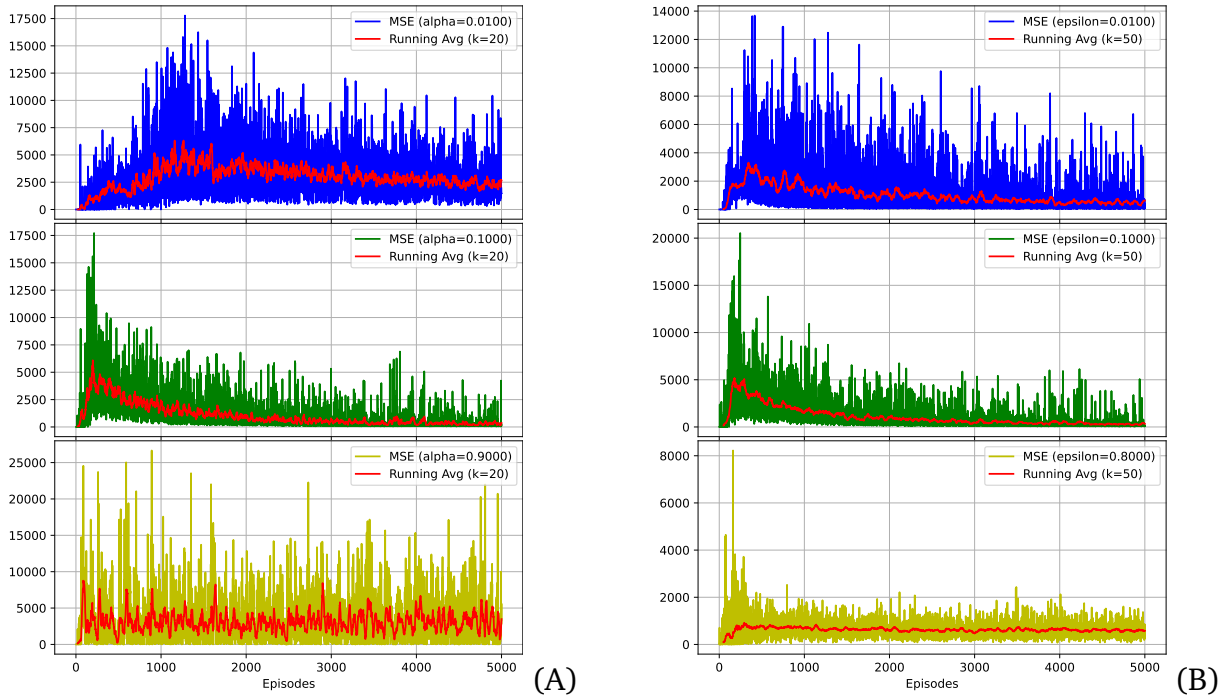
3. $\alpha = 0.90$: The MSE exhibits high oscillations and shows no signs of stabilizing, even after many episodes. This erratic behavior is indicative of a high learning rate, where the agent frequently over-adjusts its Q-values based on recent experiences, making it hard to settle on optimal policies.

ϵ values:

1. $\epsilon = 0.01$: The agent mostly exploits its current knowledge and explores less. This can lead to suboptimal policies if the agent gets stuck in a local minimum. The moving average is very similar to the one of $\epsilon = 0.10$, however the standard deviation is much higher which indicates more variability in the agent's performance over episodes. This increased variance could be a result of the agent not exploring enough to find consistent and effective strategies, causing its policy to produce fluctuating results.

2. $\epsilon = 0.10$: The agent balances exploration and exploitation, allowing it to find and converge to better policies faster.

3. $\epsilon = 0.90$: The agent explores too much, not sufficiently leveraging its current knowledge. This leads to erratic policies and less stability in learning.



(A) Fine-tuning α (B) Fine-tuning ϵ

Figure 15: Fine-tuning MC hyperparameters