Reinforcement Learning Assignment 1: Tabular Reinforcement Learning

Josef Hamelink [1]

Abstract

This paper is a submission to the first assignment of the Reinforcement Learning course at Leiden University. The basic concepts of reinforcement learning are introduced in a tabular setting. Dynamic programming is also used to highlight its differences with reinforcement learning. The problem used to illustrate the concepts in this assignment is a stochastic variation of the *windy gridworld* problem (Sutton & Barto, 2018).

1. Introduction

Reinforcement learning (RL) is a subfield of machine learning that focuses on learning to act in an environment. The actor is typically called an *agent*. The agent's goal is to maximize the expected cumulative reward it receives over time. In this assignment, the agents try to achieve this goal by minimizing the number of steps it takes to reach the goal state. The agents that have been modelled in this assignment are Q-learning (vanilla and n-step), SARSA, and Monte Carlo. To benchmark their performance, a Dynamic Programming (DP) agent is also implemented, which is guaranteed to find the optimal policy. All algorithms employ Q-tables, in which an estimate of the expected "value" of a state-action pair is stored for all possible state-action pairs.

2. Problem description

Because we are interested in the performance of various tabular model-free RL algorithms, a slightly modified version of the *windy gridworld* problem (Sutton & Barto, 2018) suits our purposes. The size of an agent's Q-table is $|\mathcal{S}| \times |\mathcal{A}|$, where $|\mathcal{S}|$ is the number of states and $|\mathcal{A}|$ is the number of actions the agent can take in each state.

[1]Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands. Correspondence to: Thomas Moerland <LIACS>.

Source code can be found on GitHub at

https://github.com/Josef-Hlink/TabularRL

In the case of this environment (Figure 1), $|\mathcal{S}| = 7 \times 10 = 70$ and $|\mathcal{A}| = 4$, so the size of the Q-table is $70 \times 4 = 280$.

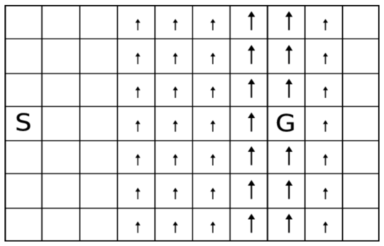

Figure 1. The environment used in this assignment.

The agent starts at the square marked with an "S" and must reach the square marked with a "G". The agent can move in all four directions, but the environment is stochastic. In columns 3[1], 4, 5, and 8, there is an 80% chance that the agent will be blown upwards by one tile, meaning that when moving right from position $(2, 3)$, the agent will probably end up in position $(3, 4)$ instead of $(3, 3)$. In columns 6 and 7, the chance is also 80%, but the agent will be blown upwards by two tiles. The agent receives a reward of $+40$ when it reaches the goal state, and a reward of -1 for any other tile it visits.

The optimal path – assuming the wind always blows when it matters, i.e. the last two steps – is shown in Figure 2.

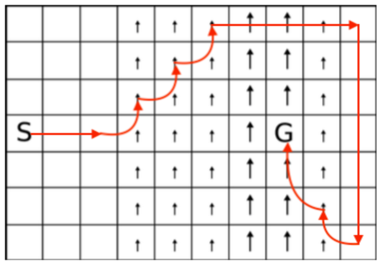

Figure 2. The optimal path to the goal state.

This path gives a total reward of $-1 \times 16 + 40 = +24$. The average reward per timestep equates to $24/17 \approx 1.412$.

[1]We begin indexing at 0, so column 3 is the fourth column to the left.

3. Methods

3.1. Dynamic Programming

The definition of Dynamic Programming (DP) comes down to solving a complex problem by breaking it down into simpler subproblems. In the case of RL, the problem is to find the optimal policy π^*, which is the policy that maximizes the expected cumulative reward. In order to solve this, we can use the Bellman equation (1).

$$V^*(s) = \max_{a \in \mathcal{A}} \left(\sum_{s' \in \mathcal{S}} p(s', r|s, a) \left(r + \gamma V^*(s') \right) \right) \quad (1)$$

Here, $V^*(s)$ is the value of state s under the optimal policy π^*, $\mathcal{A}$ is the set of actions, $\mathcal{S}$ is the set of states, $p(s', r|s, a)$ is the probability of transitioning from state s to state s' with reward r when taking action a, r is the reward, and γ is the discount factor. The way this equation leads to the optimal policy is by iteratively solving the equation for $V^*(s)$ for all $s \in \mathcal{S}$. Concretely, we initialize $V^*(s)$ to 0 for all $s \in \mathcal{S}$, and sweep over all states until the values converge. The optimal policy is then given by the choosing the action that maximizes the value of the next state, as shown in Equation 2.

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} \left(\sum_{s' \in \mathcal{S}} p(s', r|s, a) \left(r + \gamma V^*(s') \right) \right) \quad (2)$$

It is important to note that in order to take this approach, the transition probabilities must be known to the agent. This is what differentiates DP from the Reinforcement Learning algorithms that will be discussed in the upcoming sections.

3.2. Reinforcement Learning

In general, Reinforcement Learning (RL) is a framework for learning an optimal policy π^* through direct interaction with the environment. Experience is gathered by taking actions in the environment based on a behaviour policy π_b. For all intents and purposes, π_b can be just called π in this section, as no distinction will be made between the behaviour policy and the target policy. The policies that will be discussed in this section are the ε-greedy and Boltzmann (softmax) policies.

ε-GREEDY POLICY

The ε-greedy policy is a simple policy that chooses the action with the highest Q-value with probability $1 - \varepsilon$, and chooses a random (exploratory) action with probability ε. There are two main schools of thought on how to interpret ε. One is that it is the probability of choosing a random action, and the other is that it is the probability of choosing a strictly exploratory action. In this assignment, the latter interpretation will be used. The policy is described in Equation 3.

$$\pi(a|s) = \begin{cases} 1 - \varepsilon & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q(s, a') \\ \frac{\varepsilon}{|\mathcal{A}| - 1} & \text{otherwise} \end{cases} \quad (3)$$

It is clear that a high value for ε will lead to more exploration, while lower values will lead to a more greedy policy.

BOLTZMANN (SOFTMAX) POLICY

The Boltzmann policy is a more sophisticated policy that assigns probabilities to each action based on the Q-values of the actions. The probability of choosing an action is proportional to the Q-value of that action. The policy is described in Equation 4.

$$\pi(a|s) = \frac{\exp\left(\frac{Q(s,a)}{\tau}\right)}{\sum_{a' \in \mathcal{A}} \exp\left(\frac{Q(s,a')}{\tau}\right)} \quad (4)$$

The parameter τ is called the temperature, and it controls the amount of exploration. High values of τ will lead to more exploration, as the probabilities will be more uniform. As $\tau \to \infty$, $\pi(a|s)$ will approach $|\mathcal{A}|^{-1} \; \forall \, a \in \mathcal{A}$. Conversely, as $\tau \to 0$, $\pi(a|s)$ will approach 1 for the action with the highest Q-value, and 0 for all other actions.

Q-LEARNING

Q-learning is one of the most easy to implement and reason about RL algorithms. Similarly to DP, Q-learning uses the highest value of the next state to update the value of the current state. However, because it is model-free, it does not require the transition probabilities to be known to the agent. Instead, it estimates a target value (G_t) for the current state-action pair, which is based on the optimistic estimate of the value of the next state-action pair. It then updates the Q-value of the current state-action pair towards this target value. This is shown in Equations 5 and 6.

$$G_t = r_t + \gamma \cdot \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') \quad (5)$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot (G_t - Q(s_t, a_t)) \quad (6)$$

The implementation is shown in Algorithm 1.

Algorithm 1 Q-learning

```
Input: $budget, environment$
initialize $Q(s,a)$ to 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
$s \leftarrow s_0$
repeat
    sample action $a$ using Equation 3 or 4
    take action $a$ in environment to get $r$ and $s'$
    update $Q(s,a)$ with Equations 5 and 6
    if $s'$ is terminal then
        $s \leftarrow s_0$
    else
        $s \leftarrow s'$
    end if
    $budget \leftarrow budget - 1$
until $budget = 0$
```

SARSA

SARSA is a model-free RL algorithm that is very similar to Q-learning. The main difference is that it uses the action that is sampled from the behaviour policy to update the Q-value of the current state-action pair instead of using the action with the highest Q-value. One way to frame this is that SARSA uses its own behaviour policy to update the Q-value of the current state-action pair, while Q-learning uses the greedy policy for this. SARSA's update rule is identical to that of Q-learning (Equation 6), but the target value is calculated with Equation 7:

$$G_t = r_t + \gamma \cdot Q(s_{t+1}, a_{t+1}) \tag{7}$$

The implementation is shown in Algorithm 2.

Algorithm 2 SARSA

```
Input: $budget, environment$
initialize $Q(s,a)$ to 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
$s \leftarrow s_0$
sample action $a$ using Equation 3 or 4
repeat
    take action $a$ in environment to get $r$ and $s'$
    sample action $a'$ using Equation 3 or 4
    update $Q(s,a)$ with Equations 7 and 6
    if $s'$ is terminal then
        $s \leftarrow s_0$
        sample action $a$ using Equation 3 or 4
    else
        $s \leftarrow s'$
        $a \leftarrow a'$
    end if
    $budget \leftarrow budget - 1$
until $budget = 0$
```

n-STEP Q-LEARNING

n-step Q-learning is another temporal difference algorithm that is similar to Q-learning, but it uses the n-step return instead of the one-step return. The n-step return is the sum of the (discounted) rewards over the next n steps, plus a bootstrapped value of the n^{th} state-action pair. The bootstrapped value is the Q-value of the n^{th} state-action pair, which is estimated using the greedy policy (hence the name Q-learning). This means that n-step Q-learning is actually both on-policy and off-policy, assuming that the behaviour policy is not greedy. The target value is calculated with either Equation 8 or Equation 9, based on whether the episode ends before the n^{th} step or not. If the episode ends before n is reached, bootstrapping is not possible, so the second term can be omitted.

$$G_t = \sum_{i=0}^{n-1} \left[\gamma^i \cdot r_{t+i}\right] + \gamma^n \cdot \max_{a' \in \mathcal{A}} Q(s_{t+n}, a') \tag{8}$$

$$G_t = \sum_{i=0}^{n-1} \left[\gamma^i \cdot r_{t+i}\right] \tag{9}$$

Again, the same tabular update rule (Equation 6) is used to update the Q-value of the current state-action pair. The implementation is shown in Algorithm 3. It should be noted that the lines between what constitutes a single step and what constitutes a single episode became very blurry at this point in the assignment.

MONTE CARLO

Monte Carlo methods are a class of algorithms that rely on the idea of "full" playouts. This means that the agent plays out an entire episode, and then uses the rewards received during the episode to update the Q-values retroactively. This quite similar to the n-step Q-learning algorithm, but instead of using the n-step return, the full return is used. This full return can be thought of as the n-step return (Equation 9) with $n = \infty$. The implementation of Monte Carlo for this assignment is shown in Algorithm 4.

Algorithm 3 n-step Q-learning

```
**Input:** $budget, environment, n, max\_episode\_length$
initialize $Q(s,a)$ to 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
**repeat**
    $s \leftarrow s_0$
    **for** $t = 0 \ldots max\_episode\_length$ **do**
        sample action $a_t$ using Equation 3 or 4
        take action $a_t$ in environment to get $r_t$ and $s_{t+1}$
        **if** $s_{t+1}$ is terminal **then**
            **break**
        **end if**
    **end for**
    $episode\_length \leftarrow t + 1$
    **for** $t = 0 \ldots \min(n, episode\_length)$ **do**
        $m \leftarrow \min(n, episode\_length - t)$
        **if** $t + m < episode\_length$ **then**
            update $Q(s_t, a_t)$ with Equation 8
            (with bootstrapping)
        **else**
            update $Q(s_t, a_t)$ with Equation 9
            (no bootstrapping)
        **end if**
    **end for**
    $budget \leftarrow budget - 1$
**until** $budget = 0$
```

Algorithm 4 Monte Carlo

```
**Input:** $budget, environment, max\_episode\_length$
initialize $Q(s,a)$ to 0 for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$
**repeat**
    $s \leftarrow s_0$
    **for** $t = 0 \ldots max\_episode\_length$ **do**
        sample action $a_t$ using Equation 3 or 4
        take action $a_t$ in environment to get $r_t$ and $s_{t+1}$
        **if** $s_{t+1}$ is terminal **then**
            **break**
        **end if**
    **end for**
    $episode\_length \leftarrow t + 1$
    **for** $t = 0 \ldots episode\_length$ **do**
        update $Q(s_t, a_t)$ with Equation 9 ($n = \infty$)
    **end for**
    $budget \leftarrow budget - 1$
**until** $budget = 0$
```

3.3. Experimental setup

The experiments were done with a small set of hyperparameters, because I believe the main focus of this assignment to be getting a feel for the different algorithms, rather than actually finding cool results. In places where specific hyperparameters are omitted, the default values used are:

parameter	value
α	0.25
γ	1.0
ε	0.1
$max\_episode\_length$	150

The following experiments have been run:

- Q-learning with ϵ-greedy vs softmax policies:
  - ϵ-greedy with $\epsilon \in \{0.02, 0.1, 0.3\}$.
  - softmax with $\tau \in \{0.01, 0.1, 1\}$.
- Q-learning vs. SARSA:
  - $\alpha \in \{0.02, 0.1, 0.4\}$.
- n-step Q-learning with $n \in \{1, 3, 10, 30\}$ vs. Monte Carlo.

The rewards are always averaged over 50 independent runs, and the learning curves are smoothed to make the figures more interpretable. For all RL agents, a greedy evaluation run was done after each 500 timesteps to get a more complete picture of the performance.

4. Results

4.1. Dynamic Programming

In order to provide a baseline for the RL algorithms, we compute the optimal policy using dynamic programming with Q-value iteration. The optimal average reward as described in Section 2 is successfully reached following convergence of the algorithm. This can be seen in Figure 3.

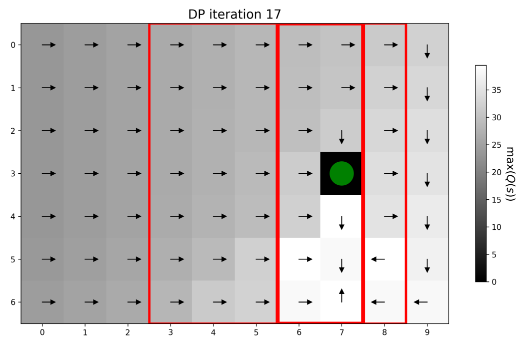

Figure 3. Policy after 17 iterations of dynamic programming.

We see that all arrows in Figure 2 (the optimal path) are also present in Figure 3, which is a good sign that the dynamic programming algorithm has found the optimal policy. Of course, there is a probability that no wind blows for the first five consecutive steps, which would make the optimal result in an average reward of $(6 \times -1 + 40)/7 \approx 5.14$. The chance of this occurring however is $0.2^5 = 3.2 \cdot 10^{-4}$, which is negligible. We can appreciate that the dynamic programming algorithm is able to find the optimal policy accounting for all possible wind interactions.

If we take a look at the process (Figures 4 & 5), we can see that the first values to be updated with any significance are the ones that are close to the goal state, and this information propagates backwards towards the initial state.

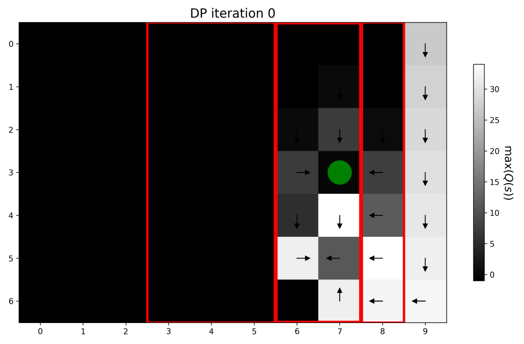

Figure 4. Dynamic Programming after the 0th iteration.

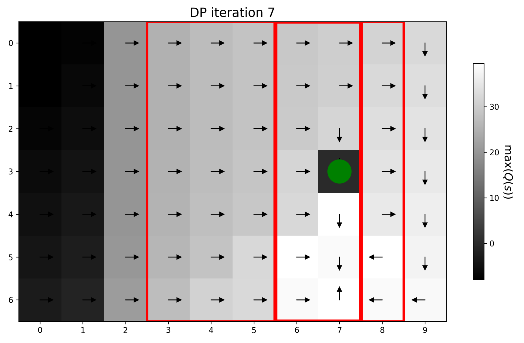

Figure 5. Dynamic Programming after the 7th iteration.

Computing $V^*(s_0)$ can be done by summing the rewards of all possible paths from s_0 to the goal state, weighted by the probability of each path. Doing this manually is tedious, so I printed $V^*(s)$ for all $s \in$ optimal path, and $V^*(s_0)$ turned out to be ± 23.312. Something that stands out is that for all (s, a, s') pairs where the transition probability is 1, $\Delta(V^*(s'), V^*(s)) = -1$, i.e. the reward for landing in that non-terminal state.

Changing the location of the goal state to $(6, 2)$ results in a very different policy, where the agent does not cross the windy columns, but instead tries to stick to the lower side of the grid. The optimal average reward is then even lower, but the path is less stable, as the wind has more of an impact on how fast the agent manages to reach the goal.

4.2. Exploration

The first experiment was to compare the performance of ϵ-greedy and softmax policies. For simplicity's sake, only Q-learning was used. The results are shown in Figure 6.

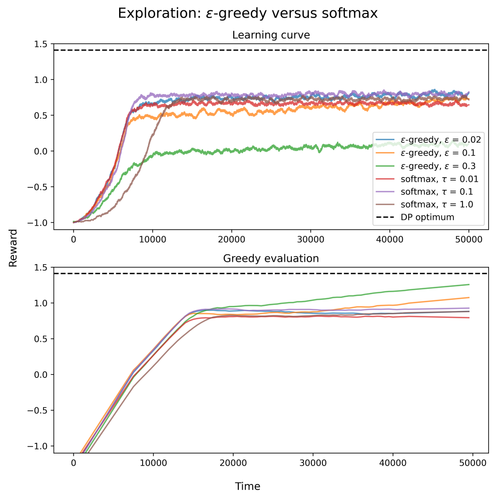

Figure 6. Performance of ε-greedy and softmax policies with Q-learning.

When looking at the learning curves, two "losers" are immediately apparent: $\varepsilon = 0.3$ and $\tau = 1.0$; the exploratory settings. The agent with $\varepsilon = 0.3$ is forced to take random actions too often, even when it has already learned a decent policy. The softmax agent with $\tau = 1.0$ is not nearly as bad, but it does take a little longer to converge than the other four agents, which seem to be very similar.

If we shift our focus to the greedy evaluation curves however, a very different picture emerges. The $\varepsilon = 0.3$ agent is actually the best performing agent, and the softmax agent with $\tau = 0.01$ is the worst. This makes sense, because the $\varepsilon = 0.3$ agent is forced to explore more, and thus has a better idea of what actions are worthwhile, even when the environment blows it off its course.

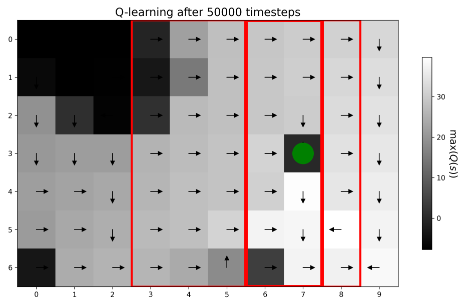

Figure 7. Policy learned by Q-learning with $\varepsilon, \alpha = 0.1$.

As can be seen in Figure 7, the resulting policy is quite similar to the DP one, but it is a bit more noisy.

4.3. On-Policy vs. Off-Policy

The next experiment was to compare the performance of Q-learning and SARSA. The results are shown in Figure 8.

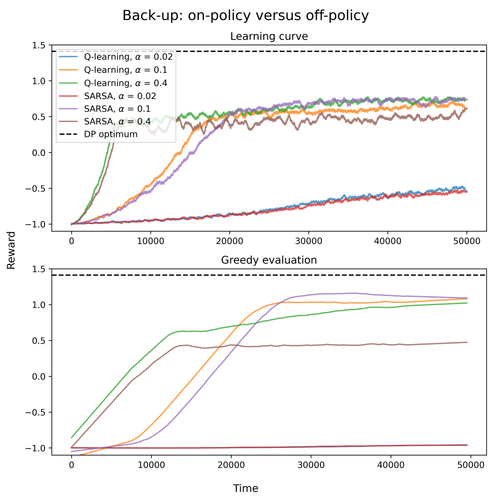

Figure 8. Performance of Q-learning and SARSA.

In general, Q-learning seems to be performing better than SARSA with the same hyperparameters, but the difference is not significant when looking at the learning curves alone. This is to be expected, because the two algorithms are very similar, and the only difference is that SARSA uses its own policy to choose the next action in the update rule, whereas Q-learning uses the greedy policy. The most notable observation to be made is that a learning rate of 0.02 is not reasonable for either algorithm, as both their learning curves as well as their greedy evaluation curves are extremely slow to converge. Personally, Q-learning with $\alpha = 0.4$ is my favorite; it leaves its SARSA counterpart in the dust when it comes to the greedy evaluation curves.

4.4. n-step and Monte Carlo

In the final experiment we compare the performance of n-step and Monte Carlo methods. The results are shown in Figure 9.

The n-step methods seem to be performing better than the Monte Carlo methods. This is because sadly, n-step methods are not viable for this problem, and Monte Carlo is basically n-step with $n = \infty$. My intuition is that this is due to the fact that these methods are not robust against stochasticity, and the wind in this problem is quite noisy. There is a very clear trend in both the learning curves and the greedy evaluation curves, where the n-step methods are performing better the lower the value of n, with $n = 1$ being the best performing method.

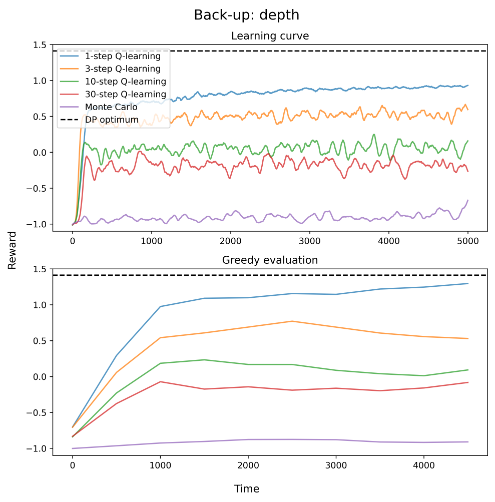

Figure 9. Performance of n-step and Monte Carlo methods.

In order to illustrate how embarrassing the performance of these n-step methods is this problem, the resulting policy after a very generous budget (algorithm has converged) is shown in Figure 10.

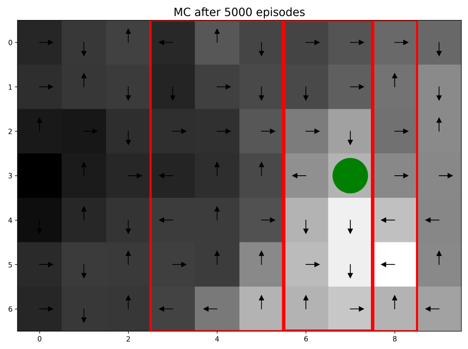

Figure 10. Policy learned by Monte Carlo.

5. Conclusion and discussion

In this report, we have discussed the implementation of several reinforcement learning algorithms and compared their performance on the windy gridworld problem. We have also shown that Dynamic Programming is the best choice of algorithm for this problem. Of course, Reinforcement Learning should not be discounted, as it is a very powerful tool for learning from experience, and it can be used to solve problems that are not solvable by Dynamic Programming, e.g. problems where no model of the environment is available. It should also be noted that both DP and RL suffer from the curse of dimensionality, but RL has a trick up its sleeve; it can employ neural networks to approximate the value function, which is a very powerful tool for solving problems with a very large (or even continuous) state space.

When deciding between ε-greedy and softmax policies, I personally lean more towards ε-greedy for this problem, as it is easier to reason about, and it is also (ever-so-slightly) more computationally efficient.

It was disappointing to see that the n-step methods performed so poorly, but I think that this is due to the fact that the problem is not well suited for these methods, and that they are not as robust against stochasticity as their single step counterparts. I do however think that it was a good idea to include them in the report, as it shows that it is not always a good idea to simply throw more compute at a problem.

References

Sutton, R. S. and Barto, A. G. *Reinformcent Learning: An Introduction*. The MIT Press, 2nd edition, 2018.