Assignment: Tabular Reinforcement Learning

Dinu Catalin-Viorel

1. Introduction

This report will compare several tabular, value-based reinforcement learning methods for sequential decision problems. Those methods compute/estimate the value of the State-Action function $Q(s, a)$ by saving old approximations in a table. The first presented method is dynamic programming, combining the ideas from reinforcement learning and planning. The key idea of this algorithm is that the agent has access to the environment's transition function and reward function without interacting with it. So it can compute values from Q-table without interacting with the environment. The other algorithms presented in this report are model-free, so they have to interact with the environment to update the Q-table. Firstly we investigated the effect of different exploration policies on the algorithms: ϵ-greedy, Boltzmann and Upper Confidence Bound[(Tijsma et al., 2016), (Saito et al., 2015)]. Secondly, we compared the on-policy methods(SARSA algorithm) and off-policy methods(Q-learning algorithm). SARSA and Q-learning use only a 1-step future reward to update the Q-table. Meanwhile, we also investigated algorithms that use multiple future rewards: the N-step Q-learning and the Monte-Carlo Algorithm.

1.1. Environment

All the methods are compared in a stochastic environment (Figure 1), Stochastic Windy Gridworld - which is an adapted version of Example 6.5 (page 130) in (Sutton & Barto, 2018)

1.2. Markov Decision Process

Usually, the problems solved by Reinforcement Learning are sequential decision problems. Markov decision processes model those sequential decision problems because they have the Markov property (the next state depends only on the current state and the actions available on it). Similar to the formalisation of the Markov Decision Process from the book (Plaat, 2022), we will define:

$$(\mathbb{S}, \mathcal{A}, T(s'|s, a), r(s, a, s'), \gamma) \rightarrow \text{Markov Decision Process} \quad (1)$$

- $\mathbb{S} \rightarrow$ finite set of legal states
- $\mathcal{A} \rightarrow$ finite set of legal actions

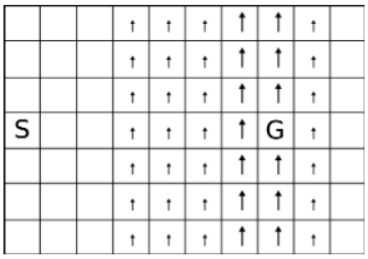

Figure 1. The environment consists of a 10x7 grid; in each cell, we have four actions, move up, down, left, right. The starting location is (0,3), indicated by 'S'. We aim to move to location (7,3), indicated by 'G'. A special feature of the environment is that there is vertical wind. In columns 3, 4, 5 and 8, we are pushed one additional step up, while in columns 6 and 7, we move up two additional steps. To make the environment stochastic, the wind does not always blow, but is randomly present on 80% of the occasions.

- $T(s'|s, a) = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a) \rightarrow$ probability that action a in state s at time t will transition to state s' at time $t + 1$
- $r(s, a, s') \rightarrow$ reward received after action a transition from state s to s'
- $\gamma \in [0, 1] \rightarrow$ discount factor

An additional important concept of reinforcement learning is policy function π, which describes how an action from state s is chosen.

$$\pi : \mathbb{S} \rightarrow p(\mathcal{A}) \quad (2)$$

$$p(\mathcal{A}) \rightarrow \text{ a probability distribution function over } \mathcal{A} \quad (3)$$

$$\pi(a|s) \rightarrow \text{ probability to take action a in state } s(s \in \mathbb{S}) \quad (4)$$

An agent that operates in the environment, samples traces(τ_t^n) (a sequence of actions, their rewards and the state after action). So, at time t, the agent is in the state s_t, it will repeat the following procedure for n times: It will choose an action a_t which moves it in a new state $s_{t+1} \sim T(s_t, a_t, s_{t+1})$ and it will get o reward of r_t after

following the action a_t in state s_t.

$$\tau_t^n = \{s_t, a_t, r_t, s_{t+1}, ..., s_n, a_n, r_n, s_{n+1}\} \quad (5)$$
$$\tau_t = \tau_t^\infty \quad (6)$$

The sum of the cumulative reward of a trace is called return(the overall reward got by the agent that follows a certain trace)

$$R(\tau_t) = r_t + \sum_{i=1}^{\infty} \gamma^i * r_{t+i} \quad (7)$$

Informally, the goal is to find the optimal policy $\pi^\star$, such that it maximises the return of τ_0(the trace starting in the initial state s_0).
We also introduce two functions:

- State value $V^\pi(s)$ This function returns the expected return value of an agent starting in the state s that follows the policy π;

$$V^\pi : \mathbb{S} \to \mathbb{R} \quad (8)$$
$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)} [R(\tau_t) | s_t = s] \quad (9)$$
$$V^\pi(s) = \mathbb{E}_{\tau_t \sim p(\tau_t)} \left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1} | s_t = s \right] \quad (10)$$

- State-Action value $Q^\pi(s, a)$ This function returns the expected return value of an agent starting in the state s choosing action a and following the policy π;

$$Q^\pi : \mathbb{S} \times \mathcal{A} \to \mathbb{R} \quad (11)$$
$$Q^\pi(s, a) = \mathbb{E}_{\tau_t \sim p(\tau_t)} [R(\tau_t) | s_t = s, a_t = a] \quad (12)$$
$$Q^\pi(s, a) = \mathbb{E}_{\tau_t \sim p(\tau_t)} \left[\sum_{i=0}^{\infty} \gamma^i * r_{t+1} | s_t = s, a_t = a \right] \quad (13)$$

Formally, the goal of reinforcement learning is to find the best policy $\pi^\star$

1. $\pi^\star(a|s) = \text{argmax}_\pi V^\pi(s_0)$
2. $\pi^\star(s) = \text{argmax}_{a \in \mathcal{A}} Q^\pi(s, a)$

Tabular Q-value algorithms compute/estimate the value of the State-Action function ($Q^\pi(s, a)$), saving for all states the values of the function corresponding to each action in a table. In the end, the optimal policy is computed from this table according to $\pi^\star(s) = \text{argmax}_{a \in \mathcal{A}} Q^\pi(s, a)$.

2. Dynamic Programming

In the first part of this report, we will investigate dynamic programming, which combines the ideas of reinforcement learning and planning. In order to use this method, the agent needs to have access to the environment's transition function and reward function without interacting directly with it. So, we consider that the following two functions for any state ($s \in \mathbb{S}$) and action ($a \in \mathcal{A}$) can be accessed by the agent.

- $p(s'|s, a) \to$ the probability to get state s' after taking the action a in the state s, this is a description of the transition function;
- $r(s, a, s') \to$ the reward received after taking the action a in state s and getting to the state s';

With access to the transition and the reward function, we can compute the best state-action value ($Q(s, a)$) based on all the possible new states the agent can reach. The expected value of taking action a in state s depends on the best possible action from s' where s' is reachable from s by tacking the action a. So, we have the following updating rule for every state-action pair:

$$Q(s, a) = \sum_{s'} \left[p(s'|s, a) \cdot \left(r(s, a, s') + \gamma * \max_{a'} Q(s', a') \right) \right] \quad (14)$$

Using this updating rule, we can modify the values for $Q(s, a)$ in no particular order until the newly computed values ($\hat{Q}(s, a)$) differ from the old values($Q(s, a)$) by a threshold (η)($|\hat{Q}(s, a) - Q(s, a)| < \eta$).

2.1. Experiments

Firstly, we want to investigate how the values change over time during the algorithm Tabular Q-value iteration (Dynamic Programming) training process. From the training process captured in figures: Figure 2, Figure 3, Figure 4, we can observe how positive values of returns propagate from the goal state to the states that point towards the goal states. We observe that for the first iteration, the pairs state-action (s, a) that get positive Q value are the ones such that taking action a in state s will get the agent to the goal state. In the second iteration, pairs that are two actions away from the goal states and so on.

From the final form of the Q table values (Figure 4), we can observe that Dynamic programming is an overachieving method, computing the expected return for any starting state-action pair. Starting from the goal state, we observe that the values Q function decreases while the number of actions needed to achieve the goal increases.

For the second part, we want to investigate how the algorithm behaves for different goal states; from figures: Figure 4 and 5, we observe that the pattern of the paths is opposite even though the end goal is diagonally adjacent. For the goal state of $[7, 2]$, the agent learns to use the wind to its advantage to achieve the final goal by taking a loop around the

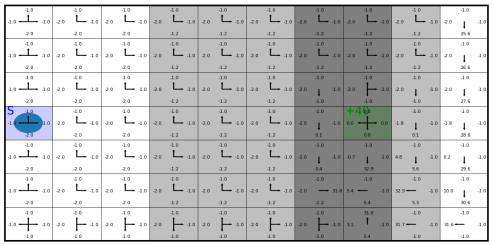

Figure 2. The test environment at the beginning of the training of the algorithm Tabular Q-value iteration (Dynamic Programming) (Alg.**??**). The goal state has the position [7,3]. Each cell state contains the corresponding $Q(s, a)$ values.

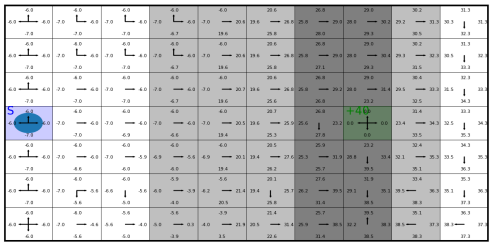

Figure 3. The test environment in the middle of the training of the algorithm Tabular Q-value iteration (Dynamic Programming) (Alg.**??**).The goal state has the position [7,3]. Each cell state contains the corresponding $Q(s, a)$ values.

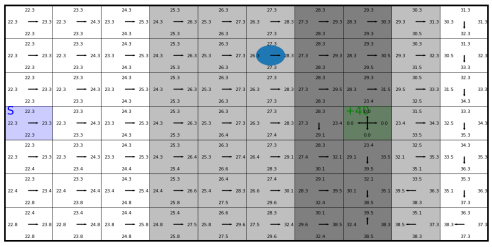

Figure 4. The test environment after the training of the algorithm Tabular Q-value iteration (Dynamic Programming) (Alg.**??**).The goal state has the position [7,3]. Each cell state contains the corresponding $Q(s, a)$ values.

goal, while for the goal state of $[6, 2]$, it is fighting against the wind to remain on the first line until he gets to the same column as the goal state. This behaviour is also seen in the standard deviation of the mean reward per time-step achieved by a greedy agent(Table 1), which is higher than the other case because for the goal state of [6,2], the wild works against the agent (the randomness of wind implies the agent wondering until the wind aligns properly).

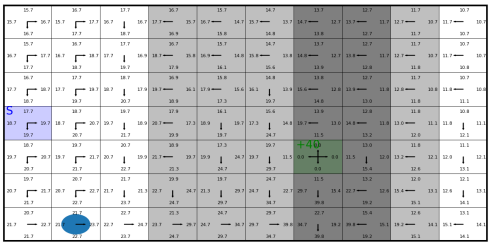

Figure 5. The test environment after the training of the algorithm Tabular Q-value iteration (Dynamic Programming) (Alg.**??**).The goal state has the position [6,2]. Each cell state contains the corresponding $Q(s, a)$ values.

Table 1. The state value for the starting position V*(s=3) and the mean reward per time-step achieved by a greedy agent(the mean is also computed over 50 repetitions) for 2 different goal states([7,2],[6,2])

GOAL STATE	MEAN REWARD PER TIME-STEP	V*(S=3)
$(7, 3)$	1.32 ± 0.12	23.307
$(6, 2)$	1.27 ± 0.78	19.748

In Table 1, we have the State value function $V*$ for starting state $s = 3$, which describes the expected cumulative reward that an agent achieves by starting in state $s = 3$.

3. Model-free Reinforcement Learning

For the following algorithms, the agent loses access to the transition and reward function without interacting with the environment. So, the agent has to interact with the environment to learn the best policy. The following algorithms estimate the values of $Q(s, a)$ based on experience gained through interacting with the environment. For those algorithms, we introduce the following value:

$G_t \rightarrow$ the new back-up estimate/target for state s_t and action a_t (it shows what should be the new value of $Q(s_t, a_t)$ based on the future taken actions and we will slowly change to it)

For the experiments in the following sections, we distinguish two different measurements:

1. One shows the reward the agent gained at every step during training. For plotting the results, we ran the training period for 50 independent runs and plotted the mean. Because there is a lot of noise caused by the different number of steps needed to reach the goal state, we also smoothed the curve.

2. One shows the mean reward per time-step achieved by a greedy agent that uses the learned form of the Q table every N number of iterations. For those measurements, we compute the mean reward per time-step every 500 iterations of training, adding a cap of at most 150 steps for the greedy agent to get to the goal state. We also computed those values for 50 independent and smoothed the resulting curve.

3.1. Exploration

In Model-free Reinforcement Learning, the agent has to interact with the environment and explore it to estimate $Q(s, a)$. In order to facilitate the exploration, there are several policies for action selection during training that can induce some randomness.

3.1.1. Action selection - policy

We have the following policies to induce some randomness in selecting the action used in a state to facilitate exploration. Without this randomness, there is a chance that the agent to hit the goal and even create a trace from the initial state to the goal state. Still, without exploration, it can't try different traces that might have better cumulative return.

- The ϵ-greedy policy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon * \frac{|\mathcal{A}|-1}{|\mathcal{A}|}. \text{ if } a = \text{argmax}_{a' \in \mathcal{A}} \hat{Q}(s, a') \\ \frac{\epsilon}{|\mathcal{A}|}, \text{ otherwise} \end{cases} \tag{15}$$

$$\epsilon \in [0, 1] \tag{16}$$

  This policy gives a high probability of choosing the best possible action based o current tabular values of $\hat{Q}(s, a)$,. And if the action is not the best possible one, it is chosen uniformly from the rest. For $\epsilon = 0$, we have a greedy policy (always choosing the best possible action), while for $\epsilon = 1$, we have a discrete uniform distribution over all actions.

- The Boltzmann policy:

$$\pi(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{\hat{Q}(s,a')/\tau}} \tag{17}$$

$$\tau \in (0, +\infty) \tag{18}$$

  This policy computes the probability of choosing an action based on the current State-Action value $\hat{Q}(s, a)$. For $\tau \to \infty$, we will have a discrete uniform over the actions ($\frac{1}{\tau} \to 0, \pi(a|s) \to \frac{1}{\sum_{a' \in \mathcal{A}} 1} = \frac{1}{|\mathcal{A}|}$).For $\tau \to 0$, we will have a discrete uniform over the actions ($\frac{1}{\tau} \to 0, \pi(a|s) \to \frac{1}{\sum_{a' \in \mathcal{A}} 1} = \frac{1}{|\mathcal{A}|}$). While for $\tau \to \infty$, we will have a greedy policy.

- Upper Confidence Bound [(Tijsma et al., 2016), (Saito et al., 2015)]
  Another method is to compute a confidence interval for each action and choose the one with the highest upper bound. So we have:

$$\pi(s) = \text{argmax}_{a' \in \mathcal{A}} \hat{Q}(s, a') + C * \sqrt{\frac{\ln(t)}{\hat{N}_t(s, a')}} \tag{19}$$

  Where:

  - $C * \sqrt{\frac{\ln(t)}{\hat{N}_t(s,a')}} \to$ confidence interval
  - $C \to$ is a constant that determine the tendency of exploration
  - $t \to$ is the current time step in training process
  - $\hat{N}_t : \mathbb{S} \times \mathcal{A} \to \mathbb{N}$, $\hat{N}(s, a) =$ the number of times the agent was in state s and chose the action a until time t.

  An action that was selected a very low number of times will have a higher confidence interval, increasing its chance to be chosen.For this policy chosen a higher value of C will increase the exploration while a lower value of C will increase the exploitation.

3.1.2. Q-learning

The first algorithm that estimates the value of the State-Action is Q-Learning. It computes the new values for $Q(s_t, a_t)$ based on the best possible action that it can choose on state s_{t+1}. This is an off-policy method because the update doesn't take into account the chosen action in state s_{t+1} and always takes into account the best possible action.

$$G_t = r_t + \gamma * \max_{a'} Q(s', a') \tag{20}$$

The updating rule:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (G_t - Q(s_t, a_t)) \tag{21}$$

$$\alpha \to \text{ learning rate} \tag{22}$$

3.1.3. Experiments

For the following experiments, we want to investigate how the action selection policy influences the learning ability of the agent. We compared ϵ-greedy softmax and UCB1 exploration with different values for ϵ temperature and C. We also compared how linear annealing on those parameters affects the learning process. From Figure 6 and Figure

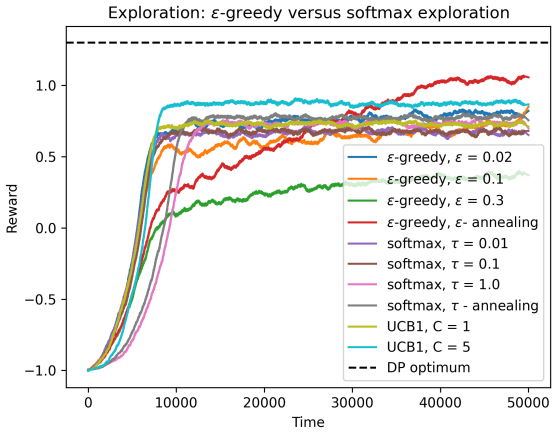

Figure 6. The rewards gained overtime by the training agent using ϵ-greedy and softmax for different values of ϵ and temperature. The algorithm used is Q-learning. The curved represents the mean over 50 repetitions, and it is smoothed.

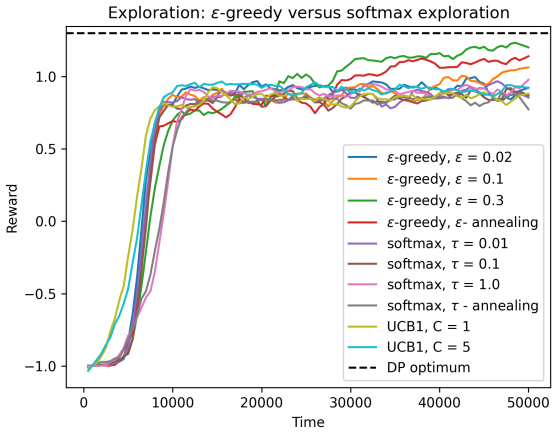

Figure 7. The mean reward per time-step gained by a greedy agent every 500 training iterations using ϵ-greedy and softmax for different values of ϵ and temperature. The algorithm used is Q-learning. The curved represents the mean over 50 repetitions, and it is smoothed

7, we can observe that all the methods generally perform equally well. Even though in Figure 6, we see that ϵ-greedy didn't outperform the others, it can't achieve a higher density of successive getting to the goal state during training. This shows that this policy is better at choosing random paths that may or may not optimise the current policy. If we look at Figure 7, this behaviour of training multiple random actions paid off because we can see that a greedy agent trained using ϵ-policy with a high ϵ value or with linear annealing can achieve higher mean reward per time-step than other methods, those values are getting closer and closer to the optimal value computed using Dynamic Programming. In contrast, the other methods converged to a sub-optimal value. The performance shown by the high value of ϵ shows that exploration plays a very important role in the training process of the gent.

3.2. Back-up: On-policy versus off-policy target

3.2.1. SARSA

Like Q-learning, the SARSA algorithm uses only one step forward to compute the new values for $Q(s_t, a_t)$; the only difference is that SARSA is On-policy (the update uses the chosen action at time $t+1$, not the best possible one).

$$G_t = r_t + \gamma * Q(s_{t+1}, a_{t+1}) \tag{23}$$

The updating rule:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (G_t - Q(s_t, a_t)) \tag{24}$$

$$\alpha \rightarrow \text{ learning rate} \tag{25}$$

3.2.2. Experiments

In this section, we want to compare the On-policy method(SARSA) and the Off-policy method(Q-learning). We will also investigate how the learning rate α affects the convergence speed of the algorithm.

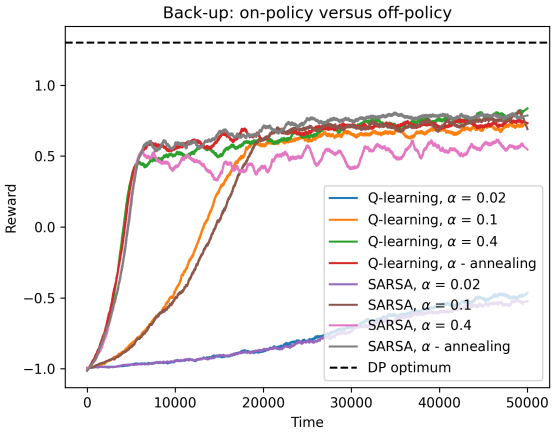

Figure 8. The mean reward per time-step gained by a greedy agent every 500 training during Q-learning or SARSA algorithms for different learning rate values (α). The curved represents the mean over 50 repetitions, and it is smoothed.

From Figure 8 and Figure 9, we observe that there is no observable difference between the On-policy method(SARSA)

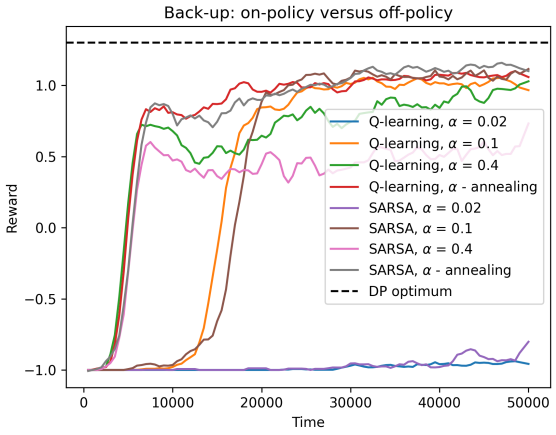

Figure 9. The rewards gained over time by the training agent using Q-learning or SARSA algorithms for different learning rate values (α). The curved represents the mean over 50 repetitions, and it is smoothed.

and the Off-policy method(Q-learning), both achieving similar performances for the same value of learning rate(α). Moreover, there is a high correlation between the value of the learning rate(α) and the performance of algorithms. For a very small α, we observe that the algorithms learn very slowly. Even though Figure 8 shows us that the agent started to achieve the goal during the training process, Figure 9 shows that a greedy agent usually fails to achieve it in 150 steps. Increasing the value of α, the algorithm converges faster to a state where the training agent finds the goal state more often. Still, for a very high learning rate, we observe in Figure 9 that the greedy agent has more noise in finding the goal state, every update changing a lot the value in the Q-table, modifying a lot the behaviour of the greedy agent.

3.3. Back-up: Depth of target

The methods presented until now use only the future return of only one step forward in time. The next algorithms use multiple forward steps to update State-Action function $Q(s_t, a_t)$. For those algorithms, we also introduce the idea of episode training; the updates of the Value-Action function will happen at the end of the episode (episodes end when it reaches a number of steps or the agent achieve the goal state).

3.3.1. N-STEP Q-LEARNING

This method uses n-step return to compute the new value for $Q(s_t, a_t)$. So, the new value depends on the cumulative reward achieved by the trace starting on state s_t and action a_t and finishing on state s_{t+n+1} and on the values of $Q(s_{t+n+1}, a*), \forall a \in \mathcal{A}$. Like Q-learning, the value used after the n step is the best $Q(s_{t+n+1}, a*)$, which makes the algorithm a mixed on-policy and off-policy method (for the first n-1 steps it counts the return of the chosen actions, not the best possible action and for the n-th action it uses the best possible one).

$$G_t = \sum_{i=0}^{n-1} \gamma^i * r_{t+i} + \gamma^n \max_a Q(s_{t+n}, a) \tag{26}$$

The updating rule:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (G_t - Q(s_t, a_t)) \tag{27}$$

3.3.2. MONTE CARLO

On the extreme part of multiple forward steps to update the tabular value of Q, we have the Monte-Carlo algorithm that uses all the state-action pairs of the episode to update the values.

$$G_t = \sum_{i=0}^{\infty} \gamma^i * r_{t+i} \tag{28}$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (G_t - Q(s_t, a_t)) \tag{29}$$

3.3.3. EXPERIMENTS

In this section, we will see the performance of the Monte-Carlo algorithm and the N-Step Q-learning for a different number of steps. We will also discuss how episode training affects the learning process. From Figure 10 and 11,

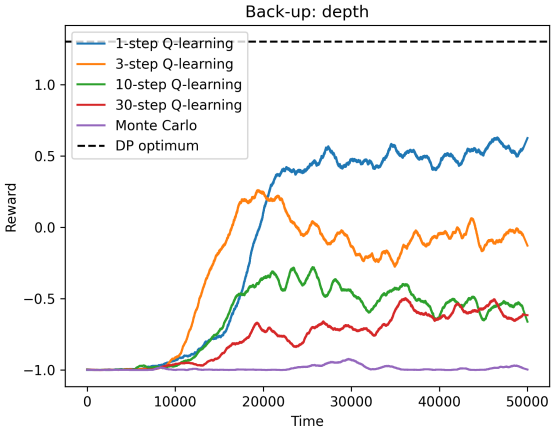

Figure 10. The rewards gained over time by the training agent using the N-step Q-learning algorithm with different values of N or using the Monte-Carlo algorithm. The curved represents the mean over 50 repetitions, and it is smoothed.

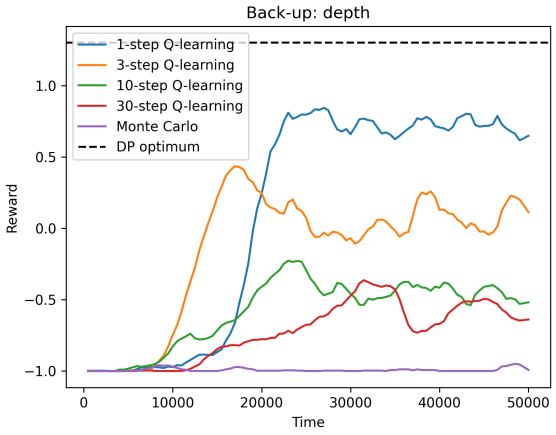

Figure 11. The mean reward per time-step gained by a greedy agent every 500 training during the N-step Q-learning algorithm with different values of N or during the Monte-Carlo algorithm. The curved represents the mean over 50 repetitions, and it is smoothed.

we observe that the Monte-Carlo algorithm failed to converge. After each training episode, the cumulative reward is propagated to all the states, so every iteration that doesn't achieve the goal state introduces a lot of random noise in the Q-table; even if the training agent achieves the goal state, the high noise in the Q-table will highly decrease the chance to create a clear path from the source to the end goal.
Moreover, for the Q-learning algorithm, we observe that episode learning reduces the learning speed and the converged value compared with the Q-learning algorithm without episode learning, this was expected because the updates are less frequent, and during an episode, the Q-table remain unchanged.
Also, increasing the number of steps of the N-step Q-learning algorithm reduces the algorithm's speed and performance.

4. Discussion

4.1. DP versus RL

Comparing Dynamic Programming with any Reinforcement Learning algorithm is complicated because their use cases differ. Dynamic programming is a model-based method, meaning it can access the environment's transition and reward functions without interacting with it. In contrast, reinforcement learning methods presented in this report are model-free, which means they don't have access(they have to learn from experience, which is the best action at any state). With this disclaimer, we can state that dynamic programming is a very powerful method that converges quickly to the best policy with the disadvantage that the transition and reward functions are available or even possible to compute without interacting with the environment. Meanwhile, we don't need any prior knowledge of the environment for reinforcement learning, making those algorithms usable in more diverse real-life applications where the transition function and reward function usually can't be computed to be used as prior knowledge.

4.2. Exploration

We investigated different exploration policies ϵ-greedy, softmax and upper confidence interval. ϵ-greedy and softmax policies showed similar results with small better results for ϵ-greedy because it facilitates using new routes through the uniform distribution of non-best actions. We also introduce another exploration policy that takes into account the number of times an action was taken by an actor in a state during training.

4.3. Back-up, on-policy versus off-policy

From the observation from this report, there is not a high difference between on-policy and off-policy, but there are some pros and cons to both methods:

1. On-policy
    - it updates/improves the policy that was used to make decisions
    - it may get stucked in local minima
2. Off-policy
    - it updates/improves one policy while following another
    - it is more flexible if new routes appear

We consider N-step Q-learning a mixed on-policy and off-policy method because, to compute the new value of $Q(s_t, a_t)$ for the first n-1 steps, it uses the return of the chosen actions(on-policy) and for the n-th step it uses the best possible one(off-policy.

4.4. Back-up, target depth

This report studied the 1-step, n-step and Monte Carlo methods. For this particular task, the 1-step methods outperform any other algorithm. The information is transmitted faster by n-step/Monte Carlo algorithms.
The Monte Carlo algorithm relies on the full trajectories of an agent acting within an episode of the environment to compute the cumulative reward. Because the learned policies are stochastic, it leads to variance in the reward received in a trajectory, so Monte Carlo presents a very high variance. Meanwhile, the 1-step method shows a small variance because every update is computed using the immediate

reward and the estimates of the next step, but it shows a bias because it won't find the actual cumulative reward and it will approximate it using an old estimation. Somewhere between those methods are n-step algorithms that try to find a balance between the high variance and no bias of Monte Carlo and the bias and low variance of the 1-step method. The best method to use depends on the task; the best balance between variance and bias might differ from task to task (Arthur, 2018). Because of that, we expect that the method that will converge to optimal policy is the one that achieves a good enough balance between variance and bias.

4.5. Curse of dimensionality

In this report, we investigate different tabular value-based methods. One of its biggest disadvantages is the necessity to save a table of all possible state-action pairs. The number of those pairs can explode quickly with the dimension of the environment, making the curse of dimensionality one of the biggest problem of these methods. To overpass this issue, we won't save the value Q function in a table; we will try to approximate its value by different methods such as Neural Networks, Decision Trees etc.

References

Arthur, J. Making sense of the bias / variance trade-off in (deep) reinforcement learning, January 2018. URL https://blog.mlreview.com/making-sense-of-the-bias-variance-trade-off-in-deep-reinforcement-learning-79cf1e83d565. [Online; posted 31-January-2018].

Plaat, A. *Deep Reinforcement Learning*. Springer, 2022.

Saito, K., Notsu, A., Ubukata, S., and Honda, K. Performance investigation of ucb policy in q-learning. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 777–780. IEEE, 2015.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Tijsma, A. D., Drugan, M. M., and Wiering, M. A. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–8. IEEE, 2016.