ASSIGNMENT 1: TABULAR REINFORCEMENT LEARNING

Giulia Rivetti
s4026543

1 INTRODUCTION

This assignment delves into the basic principles of tabular, value-based reinforcement learning, with a focus on dynamic programming [2], a bridging method between RL and planning that assumes full access to a model of the environment, and model-free RL, where there's no more access to a model. We analyze some important aspects of model-free RL, such as the importance of exploration [3, the type of policy used for back-up [4] and the depth of the backup [5]. We also compare the performance of the proposed strategies, to see which methods will outperform the others.

2 DYNAMIC PROGRAMMING

Dynamic programming is characterized by the presence of a model of the environment given as a Markov Decision Process (MDP), which is used by a given algorithm to compute the optimal policy (Sutton & Barto, 2018). In particular, we are interested in the implementation of the Tabular Q-value iteration algorithm, which sweeps through all state-actions pairs and updates the estimate of a state-action value $Q(s, a)$ (Moerland)based on the Q-iteration update following Bellman equation:

$$\hat{Q}(s, a) \leftarrow \sum_{s'} [p(s'|s, a) \cdot (r(s, a, s') + \gamma \cdot max_{a'} Q(s', a'))] \tag{1}$$

This step is followed by the update of the maximum error Δ, which represents the maximum change in Q-values across all state-action pairs during each iteration. Equation 2 shows this update:

$$\Delta \leftarrow max(\Delta, |x - \hat{Q}(s, a)|) \tag{2}$$

The algorithm converges and terminates when the maximum error Δ is below the threshold γ, outputting the optimal value function $Q^*(s, a)$ and the corresponding optimal policy $\pi^*(s)$.

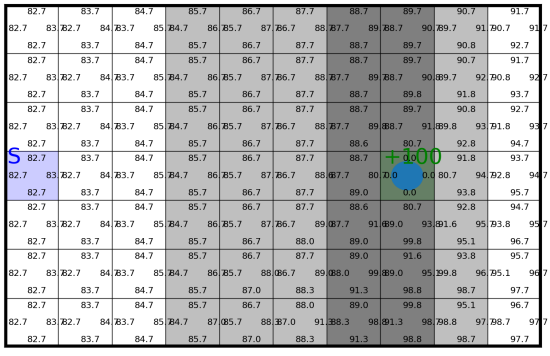


Figure 1: The plot displays the environment at the end of the execution of the Tabular Q-learning algorithm, showing the Q-value for each state-action pair. The best path to reach the goal, i.e. the optimal policy, selected by the agent is shown by the highest value of the estimated reward in each cell of the grid.

The results of the Q-value update can be seen in Figures 1, 2 and 3, which show the evolution of the Q-values respectively at the end, mid-way and beginning of the execution. Since the state-action value Q represents the estimated average return we expect to achieve when taking action a in state s, the plot at the end of the execution (Figure 1) basically shows us the path that the agent chose (hence the policy) by selecting each time the direction with the highest expected reward. In Figure 2, representing the estimates halfway through the execution of the algorithm, we can notice that for many state action pairs the Q-values are already converging toward more accurate estimates of the expected cumulative final reward. However, we can also see some low values, indicating that the agent needs to gain more knowledge of the environment in order to compute the Q-values and find the best policy. At the beginning of the algorithm (Figure 3), the agent doesn't have any knowledge at all about the environment, and therefore the state-action values are randomly initialized to low values.

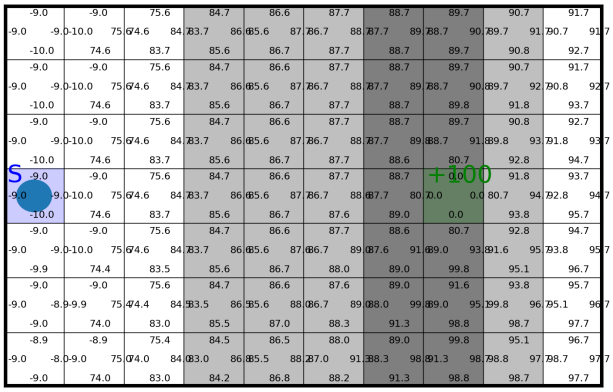


Figure 2: The plot displays the environment halfway through the execution of the Tabular Q-learning algorithm, showing the Q-value for each state-action pair. The agent has gained already enough knowledge to get more accurate estimates of the expected final rewards, but still needs more iterations to get a full knowledge of the environment.

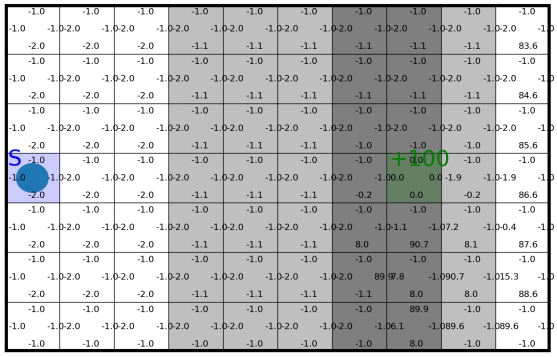


Figure 3: The plot displays the environment at the beginning of the execution of the Tabular Q-learning algorithm, showing the Q-value for each state-action pair. Here the Q-values are set to random low values because the agent has not gained knowledge about the environment yet.

We now turn our attention to the converged optimal value at the start, which represents the optimal expected cumulative future reward achievable under the optimal policy starting at a given position. Since the objective or Reinforcement Learning is to achieve the highest possible average return from the start state (Plaat, 2023), by optimal value we mean the maximum. The converged optimal value can be computed according to the following formula (Lambert, 2020):

$$V^*(s) = max_a Q^*(s, a) \tag{3}$$

Since we are interested in the value at the start, we just need to substitute into the formula the starting position, namely $(0, 3)$. The result that I have obtained is the following:

$$V^*(s = 3) \sim 82.7153 \tag{4}$$

Therefore this value tells us that expected cumulative future reward the agent is expected to gain by following the optimal policy and starting from position $(0, 3)$.

An important value that can indicate the quality of the performance of our algorithm, is the average reward per timestep. In order to compute it, we have considered 10000 iterations, because the learning process is stochastic and therefore the results can vary from one run to another. To take into account this, it's better to perform different runs and average their results. In particular, for each episode the algorithm selects an action based on the optimal policy and receives from the environment the corresponding state and reward. This is done until the episode ends, and every

time the reward is accumulated, obtaining at the end of the episode a final return. At this point, the reward per time step can be computed with the following formula:

$$r_{timestep} = \frac{R}{100 - R} \tag{5}$$

where $r_{timestep}$ is the reward per timestep and R is the final cumulative reward. As the formula indicates, to obtain the reward per timestep one just needs to divide the total reward by the number of timesteps, which can be obtained by subtracting the final reward (100 in our case) and the total reward. As already explained before, we compute this value for 10000 episodes to get the average and we end up with the following value:

$$\bar{r}_{timestep} = 5.135587341721792 \tag{6}$$

The fact that the obtained value is not particularly high is due to the negative reward of the agent at each step (-1). However, since it is a positive value we can conclude that the optimal policy is working properly.

In the given environment the goal state ($s = 52$) is terminal, meaning that when we reach it, the task ends and therefore we cannot select a new action, or transition out of that state (Moerland). Even in the presence of a terminal goal state, the algorithm is still able to converge, because when the Q-values are updated for the goal state, the rewards are set to 0, and the transition probabilities are such that they keep the agent in the same state with probability 1. In file $Experiment.py$ this is done in the following way:

Algorithm 1: Code showing how the terminal state is handled

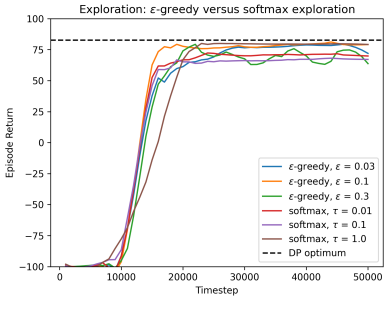
```
if state_is_a_goal then;
    p_sas[s, a, s] = 1.0;
    r_sas[s, a, ] = np.zeros(self.n_states) ;
end
```

Then, in my implementation of dynamic programming, Algorithm 1 was used in order to handle the goal state as a terminal state during the update step. Another way to solve this issue could be to consider the goal state as a terminal state explicitly and excluding it from the update with an *if done* statement.

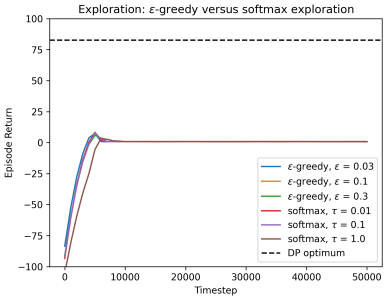
In this part of the assignment I have changed the goal state to be at position $(6, 2)$ to observe any change with respect to the previous case where the goal was at position $(7, 3)$. By running the program with this configuration it can be immediately noticed that the optimal policy performs worse than in the original case. Not only the algorithm takes much more iterations to converge (131 against the 17 iteration needed before), but also the converged optimal value at the start is much lower ($V^*(s = 3) \sim 66.88$) and so is the average reward per time-step, which in this case is only around 2.52. These results clearly show that the selection of the goal state's position can greatly impact the learning process of the agent.

3 EXPLORATION

In the model-free setting, there is no access to the model in order to favour exploring new actions in the environment. To implement tabular Q-learning in this specific setting, first I have implemented the action selection procedure, which is based either on the ϵ-greedy policy or on the Boltzman (or softmax) policy. In the former case, a random action is selected with a small probability equal to ϵ, otherwise a greedy action is taken. In the latter case, instead, the policy gives a higher probability to actions with a higher current value estimate, but still ensures exploration of other actions than the greedy one (Moerland). After selecting an action under one of those two policies, the agent takes that action and receives from the environement the corresponding next state and reward. Like in the previous case of Dynamic programming, also here the Q-values are updated. This however is done differently with respect to the previous settings: the new back-up estimate G_t is computed using



(a) This figure shows the environemnt with a single goal at position (7, 3) with a reward of +100.



(b) This figure shows the environemnt with a goals: one at position (7, 3) with a reward of +100 and the other at position (3, 2) with a reward of +5.

Figure 4: The two plots show the episode return value as a function of the number of timesteps for both ϵ-greedy and softmax policies with $\epsilon = 0.03, 0.1, 0.3$ and $\tau = 0.01, 0.1, 1.0$ respectively. The DP optimum obtained before is also shown for comparison.The learning curves were obtained by performing 20 repetitions.

the current reward and the Q-value estimates of the next state-action pair as showed in the formula below:

$$G_t = r_t + \gamma \cdot max_{a'} \hat{Q}(s_{t+1}, a') \tag{7}$$

Then, the Q-value estimates are finally updated following the tabular learning update:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot \Big[G_t - \hat{Q}(s_t, a_t) \Big] \tag{8}$$

Figure 4a shows the graphs of the episode return as a function of the number of timesteps using the ϵ-greedy (with $\epsilon = 0.03$, 0.1 and 0.3) and sotfmax (with $\tau = 0.01$, 0.1 and 1.0) policies.
By looking at the three learning curves of the ϵ-greedy policy, we can notice that for a smaller value of ϵ ($\epsilon = 0.03$) the learning curve is more stable with respect to the plots where ϵ set to 0.1 and 0.3: indeed, when we increase the amount of exploration, the agent switches back and forth between exploration and exploitation, resulting in a more unstable learning curve. Moreover, with higher values of ϵ (0.1 and 0.3) the strategy is able to reach higher episode reward values: by exploring more, the strategy has a higher change of finding new optimal actions. The main difference between the $\epsilon = 0.1$ and $\epsilon = 0.3$ learning curves is the learning speed: when ϵ is set to 0.1, the algorithm is learning faster and thus the optimal parameter value for the ϵ greedy strategy is 0.1.
As for the softmax policy, Figure 4a clearly shows that, when τ is higher ($\tau = 1$), the learning curve converges to a higher return episode value. Indeed when the temperature is higher, we ensure more exploration, allowing the agent to find better actions and reach a higher episode return. However this increased exploration results also in a slower convergence: the faster learning can be observed when τ is only 0.01. Overall, if we prioritize obtaining a higher episode return value over a faster learning, $\tau = 1.0$ would be a better choice.
Between the softmax and ϵ-greedy strategies, there's no clear winner: they both have advantages and drawbacks and the choice of the best policy depends on what we want to prioritize, whether we prefer a method that learns faster or one that reaches higher episode return values.
Figure 4a also shows how the reinforcement learning strategies perform compared to dynamic programming. The expected result, was to see that the RL strategies could not reach the maximum set by DP, since dynamic programming strategies always guarantee to converge to the optimal solution when a known and accurate model is given, whereas RL methods do not have this guarantee. However, due to the stochasticity of our environment this is not always the case and it may happen to see in a few runs that the RL strategies are actually able to reach higher values of episode return (DPa, 2023).

In our environment, the goal state can be found at position (7, 3) with a reward of +100. Now we add a second goal at position (3, 2), giving a reward of +5; to see how the performance has been affected with respect to the single goal case, we plot the same learning curves as in Figure 4a. In

the case with two goals (Figure 4b), we can notice that the performance has been hugely affected by the addition of a new goal: now all the learning curves converge at an episode return value of 0, which makes sense since the newly added goal has a reward much lower than the first goal. With two goals, however, the learning process appears to be faster: having more goals encourages the agent to explore various paths, leading to a more comprehensive understanding of the environment and potentially faster learning.
The addition of a second goal also affects the exploration-exploitation trade-off. When the environment was characterized by a single goal, exploration could be observed in the first stages of the learning process (Figure 4a); instead, with an additional goal state, there's an immediate exploitation: since the first goal offers a significantly higher reward than the other, the agent has initially prioritized exploiting this high-reward goal, rather than exploring other options. Thus now exploitation is favored more than when there was a single goal.
Moreover, now there is not much difference in performance between different values of ϵ and the best τ value is not 1.0 anymore: with this value now the learning curve reaches a lower episode return value and the strategy is also learning more slowly.

4 On-policy versus off-policy target

In this part we focus on the implementation of SARSA, an on-policy algorithm that, as such, it updates the policy with the action values of the policy. This means that during training, SARSA selects an action, evaluates it in the environment, and follows the actions, guided by the behavior policy. Therefore it samples the state space following the behavior policy, and improves the policy by backing up values of the selected actions. In particular, SARSA updates its Q-values using the Q-value of the next state s and the current policy's action (Plaat, 2023), as indicated by the formula below:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{9}$$

We now compare the learning curves of on-policy SARSA and off-policy Q-learning for different learning rates (α = 0.03, 0.1 and 0.3) (Figure 5). From the figure we can see that for $\alpha = 0.03$ even if both methods are characterized by a slow learning, during the last timesteps SARSA outperforms Q-learning, probably due to the poor performances that can characterise Q-learning in the initial part of the learning phase (Saxena et al., 2023). When the learning rate is set to 0.1, both learning curves show many fluctuations, but Q-learning can reach a higher episode return. If we increase α to 0.3 both strategies show a more stable behavior, even if SARSA appears to have more fluctuations than Q-learning, indicating that the strategy is switching back and forth between exploring and exploiting already gained knowledge. For our task, I would prefer Q-learning over SARSA because it allows to reach a high episode return, while still having a more stable behavior, hence a better exploration-exploitation balance. However, SARSA is preferable to Q-learning in situations where it's important to learn and follow the same policy during learning and execution. One common scenario is when the agent interacts with the environment in real-time (online setting), and the consequences of its actions are immediately observed (Saxena et al., 2023).

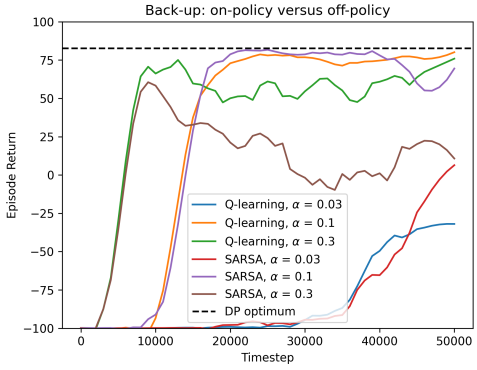


Figure 5: The plot shows the episode return value as a function of the number of timesteps of Q-learning and SARSA strategies for different value of learning rate ($\alpha = 0.03, 0.1, 0.3$). The DP optimum obtained before is also shown for comparison. The learning curves were obtained by performing 20 repetitions.

5 BACK-UP: DEPTH OF TARGET

Up to now we have only analyzed 1-step methods, which bootstrap a value estimate after one transition [(Moerland)]. However, it is also possible to sample a few steps at a time before using the reward values (Plaat, 2023); this method is called n-step. In this assignment we focus in particular on n-step Q-learning, which computes the target as follows:

$$G_t = \sum i = 0^{n-1} \gamma^i \cdot r_{t+1} + \gamma^n \cdot max_a Q(s_{t+n}, a) \tag{10}$$

Then the target is used for the standard tabular update that can be seen in formula 8.
Besides n-step Q-learning, we have also implemented Monte Carlo Q-learning, which basically randomly samples full episodes: bootstrap is omitted and the algorithm simply sums all rewards up to the end of the episode (Plaat, 2023). In this case, the target G_t is computed as follows:

$$G_t = \sum i = 0^{n-1} \gamma^i \cdot r_{t+1} \tag{11}$$

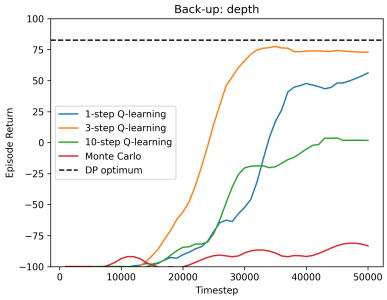
Then the Monte Carlo update is completed by simply using again the standard Q-learning update reported in formula 8.

After having implemented both n-step and Monte Carlo I have run the algorithms for 50001 timesteps; the resulting plots are showed in figure 6a. The figure shows the learning curves of Monte Carlo Q-learning and n-step Q-learning for $n = 1, 3$ and 10. The optimum value previously obtained during the dynamic programming part is also shown for comparison. By looking at the picture, we can see that n-step clearly outperforms Monte Carlo strategy: by performing bootstrapping, n-step balances the bias-variance trade-off; however, Monte Carlo doesn't use bootstrapping, resulting in high variance and in our environment also in worse performance. As for which value of n is best in the n-step algorithm, from the plots we can observe that 3-step q-learning outperforms the other strategies: for $n = 1$ the exploration is slower because we use the reward values after having sampled each step, which requires more time than using those values only after having sampled three steps. The case with the worst performance is when $n = 10$, where n has been set to a value too high for the algorithm to properly learn the optimal policy.
Since we have implemented both tabular q-learning and tabular n-step q-learning with n=1, we have plotted both learning curves to understand if the two strategies are equivalent or not. From the plot we can clearly see that they are different: tabular Q-learning shows a faster learning, however the figure suggests that if we had considered more timesteps, the n-step approach would have probably reached a higher episode return value. This difference makes sense since by looking at the code, it can be seen that the two algorithms are actually different. In tabular Q-learning, the Q-values are updated using the update rule shown in formula 7, where the target Q-value is calculated based on the immediate reward and the maximum Q-value of the next state. On the other hand, in 1-step Q-learning, the Q-values are updated based on the sum of rewards for a single step ahead, using the Bellman equation with a depth of one step 10. Moreover, 1-step Q-learning explicitly accounts for the rewards and Q-values only one step ahead, which may result in a slower learning process. In contrast, tabular Q-learning updates its Q-values based on the estimated cumulative reward from the current state to the end of the episode, which can lead to more efficient learning.

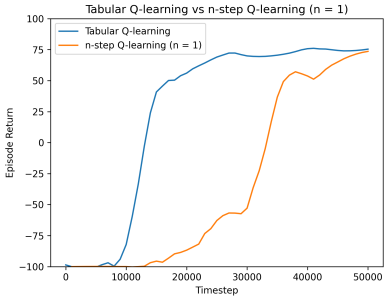
6 REFLECTION

DP versus RL Comparing Dynamic Programming and Reinforcement Learning, we can identify one main advantage of using DP over RL: DP always guarantee to converge to the optimal solution when a known and accurate model is given; this can't be guaranteed in RL settings. However since DP requires having a model and it may not always be possible to obtain a complete and correct model of the environment, especially for large and complex problems (DPa, 2023).

Exploration Between ϵ-greedy and softmax exploration methods, the latter is probably preferable, because, by selecting actions with probabilities proportional to their current values, the softmax strategy takes into account the existence of better and worse actions; instead when the ϵ greedy policy chooses the random action instead of the greedy one, it considers them to be equally good, but some actions may be worse than others and this may result in lower performance [Exchange (2020)]. On the other hand, softmax policy selects the random actions with probabilities proportional to their current values and in this way it actually takes into account the existence of better and worse actions (Exchange, 2020).



(a) This figure shows the learning curves of Monte Carlo and n-step Q-learning (with $n = 1, 3, 10$). The DP optimum is also shown for comparison.



(b) The figure depicts the learning curves of of tabular q-learning and 1-step q-learning, underlying the fact that the two algorithms are different.

Figure 6: The two plots show the episode return value as a function of the number of timesteps using the ϵ-greedy policy (with $\epsilon = 0.05$) and learning rate $\alpha = 0.1$. The learning curves were obtained by performing 20 repetitions.

A better method to explore could be using Thompson Sampling, a probabilistic algorithm that models each arm's reward distribution. It samples from these distributions to make decisions, naturally balancing exploration and exploitation by adapting to the observed rewards (Minai, 2023). Because of this, it handles exploration more efficiently than both ϵ-greedy and softmax methods.
Now we decide to set the reward for each step to 0 instead of -1, while still keeping the reward of the goal state to +100. Also in this case, we have plotted the episode return as a function of the number of time steps for three different values of ϵ (0.03, 0.1 and 0.3) for the ϵ-greedy policy, and three different τ values (0.01, 0.1 and 1) for the softmax strategy. From these plots (Figure 7), we can see that the performance of all strategies has been affected by this change: now the agent is not penalized anymore at each step, and learning can become more difficult. Indeed, in this case the learning process appears to be slower, due to the fact that the reward landscape is sparse (i.e. flat): only the final goal returns a non-zero reward. Therefore, in order to fill the Q-values with actionable information on where to find the goal, first the algorithm must be lucky enough to choose a state next to the goal, including the appropriate action to reach the goal. Only then the first useful reward information is found and the first non-zero step towards finding the goal can be entered into the Q-function (Plaat, 2023). Moreover in environments with sparse rewards the exploration-exploitation trade-off becomes also more difficult to balance and may result, as we observe in our case, in reaching smaller episode return values.

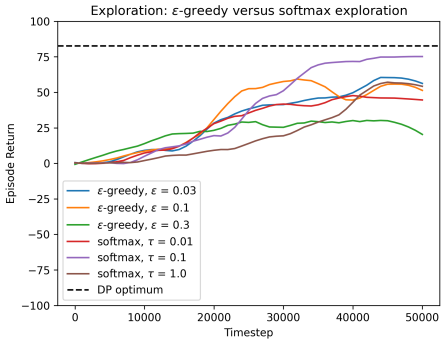


Figure 7: The plot shows the episode return value as a function of the number of timesteps for both ϵ-greedy and softmax policies with $\epsilon = 0.03, 0.1, 0.3$ and $\tau = 0.01, 0.1, 1.0$ respectively, when the reward per step is 0.The learning curves were obtained by performing 20 repetitions.

Discount Factor In this part of the assignment we have decided to keep the reward at each step to 0, while the reward for the goal state is again +100. With these settings, we try to change the value of the discount factor γ from 1.0 to 0.99 to see which approach is able to find the shortest path. When γ is set to 1.0, it doesn't have any effect at all; however when its value is decreased, the discount factor reduces the impact of far away rewards (Plaat, 2023). In our case, when the discount factor is set to 1 and the agent is not penalized by taking a step, the policy doesn't actually distinguish between a longer and a shorter path to reach the goal: the final reward for both cases will still be 100, because each step is neither decreasing or increasing the total reward. Indeed the total reward is computed by just adding up the reward at each step but since each steo has a reward of 0 now and the final

goal has a reward of +100, the total final reward for any path will be +100. However, when γ is set to 0.99, the discount factor is actually introducing a penalization for the agent. In this case the total reward is not just given by the sum of the reward at each step taken, but we also need to take into account the discount factor, as shown by the following formula:

$$total\ reward = r_{goal} \cdot \gamma^{\#steps\_taken} \tag{12}$$

where r_{goal} is 100, γ is 0.99 and $\#steps\_taken$ indicates the number of steps needed in order to reach the goal state for a given path. Since now the total reward is updated using the above formula, a longer path would get a smaller reward with respect to a shorter one. Therefore, it can be concluded that the approach using a discount factor of 0.99 will find the shortest path to the goal; however this doesn't happen when γ is set to 1 and the agent is not penalized in any way.

Back-up, on-policy versus off-policy In on-policy learning, such as SARSA, the learning takes place by using the value of the action that was selected by the policy; instead in off-policy learning, the learning takes place by backing up values of another action, not necessarily the one selected by the behavior policy like the on-policy learning does. This is what happens when the Q-learning strategy is used. An advantage of Q-learning is that it converges to the optimal policy, since it learns from the greedy rewards. Instead, on-policy methods such as SARSA, converge only when we use a variable ϵ-policy in which ϵ goes to zero (Plaat, 2023). Moreover on-policy methods are known to be relatively less sample-efficient overall on account of relying on a single policy for both exploration and exploitation. However, adn advanatage of on-policy strategies like SARSA is that they tend to have lower computational complexity since only a single policy needs to be stored and updated at every step; for the same reason they are also easier to tune in general.

Back-up, target depth The bias-variance trade-off is determined by the use of bootstrapping. Monte Carlo doesn't use bootstrapping, resulting in low bias and high variance (since the action choices are fully random). 1-step Q-learning uses bootsrapping and thus it quickly refines Q-function values with rewards after each step, leading to lower variance but higher bias due to the persistence of old reward values in the updated function values. Between these two strategies we can place n-step methods, which is characterized by medium bias and medium variance (Plaat, 2023).
An advantage of 1-step methods is that they update value estimates after each individual step, allowing for faster learning and adaptability to changing environments. However, as we have already noticed, they have the disadvantage of having a high variance. N-step methods are able to balance the bias-variance trade-off, but the introduction of a new parameter (n) requires tuning and can affect performance. Monte Carlo has the advantage of being a simple approach; however it may not be sample-efficient, since a full episode has to be sampled before the reward values are used. For the task and environment that we have considered,between these three strategies I would prefer n-step Q-learning. As we have already observed before in Figure 6a, n-step outperforms both Monte Carlo and 1-step for the reasons that I have already explained while analyzing their plots.
One-step methods propagate information the fastest because they update value estimates after each individual step. This allows them to quickly incorporate new experiences into the learning process. Monte Carlo (MC) methods may converge on the optimal policy because they use the full trajectory to update value estimates, leading to unbiased estimations of state values.

Curse of dimensionality However tabular RL methods work only for small problems: when the input space is high dimensional, we cannot store the entire state space (all possible pixels with all possible values) as a table. Therefore tabular methods are characterized by an exponential need for observation data as the dimensionality grows, a problem that takes the name of curse of dimensionality. The curse of dimensionality states indeed that the cardinality (number of unique points) of a space scales exponentially in the dimensionality of the problem (Plaat, 2023). Since they don't maintain a large table of state-action values, deep learning methods can overcome the curse of dimensionality problem by approximate the value or policy functions directly from raw state inputs.

REFERENCES

How can dynamic programming be used for reinforcement learning? https://www.linkedin.com/advice/0/how-can-dynamic-programming-used-reinforcement, 2023.

Artificial Intellige Stack Exchange. What is the difference between the e -greedy and softmax policies? https://ai.stackexchange.com/questions/17603/what-is-the-difference-between-the-epsilon-greedy-and-softmax-policies, 2020.

Nathan Lambert. Convergence of reinforcement learning algorithmsg, 2020. URL https://towardsdatascience.com/convergence-of-reinforcement-learning-algorithms-3d917f66b3b7.

Yuki Minai. Exploring multi-armed bandit problem: Epsilon-greedy, epsilon-decreasing, ucb, and thompson sampling. https://medium.com/@ym1942/exploring-multi-armed-bandit-problem-epsilon-greedy-epsilon-decreasing-ucb-and-thc 2023.

Thomas Moerland. *Continuous Markov Decision Process and Policy Search*.

Aske Plaat. *Deep Reinforcement Learning*. 2023.

Vidit Saxena, Burak Guldogan, and Doumitrou Daniil Nimara. On-policy and off-policy reinforcement learning: Key features and differences. https://www.ericsson.com/en/blog/2023/12/online-and-offline-reinforcement-learning-what-are-they-and-how-do-they-compare, 2023.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.