Assignment: Tabular Reinforcement Learning

Maria Ieronymaki - s3374831[1]

Abstract

This document reports the basic concepts of Tabular Reinforcement Learning. It considers model-based methods such as Dynamic Programming and model-free methods such as Q-learning and SARSA. It tackles the issue of on and off policy learning and compares them. Finally introduces a depth in the target function in order to implement n-step Q-learning and Monte Carlo methods.

1 Introduction

The popularity of reinforcement learning has increased tremendously in recent years and we have now reached the point where we are able to achieve amazing things with it. This document reports all the methods that represent the basic principles in a value-based tabular reinforcement learning. Specifically, it covers the topic of Dynamic Programming which is considered a method that stands in between planning and reinforcement learning. In this context, we are aware of the model and in particular of the transition probabilities and the rewards for each state s and action a. It also highlights the method's strengths and disadvantages and complexities such as the curse of dimensionality. We then move on to the Model-free topic where we do not have access to the model where specifically off-policy methods (Q-learning) and on-policy methods (SARSA) are compared. It then concludes with the implementation of n-step methods using a depth target function such as the Monte Carlo method. For all the methods implemented, the results and their interpretations are reported based on the tuning of the parameters made. All experiments were tested on the same environment called Stochastic Windy Gridworld.

2 Environment

The environment used for the implementation of the methods is called Stochastic Windy Gridworld (see Fig.1) (Sutton & Barto, 2018). It is a 10x7 grid, where each cell is numbered from 0 to 70 starting from the bottom left cell and moving upwards until reaching the seventieth cell located at the bottom right. The agent moves up, down, left and right and his initial position indicated as 'S' in the figure is (0,3). The goal is to move the agent to the final position (7,3), indicated as 'G'. In the environment and especially in columns 3,4,5,8 there is a wind that makes the agent move one step upwards while in columns 6 and 7 the wind pushes it upwards by two steps. The fact that the presence of the wind is random, since it blows 80% of the time makes the environment stochastic. The agent's reward is -1 in each step and +35 if the goal is reached. Achieving the final state leads to the termination of the episode.

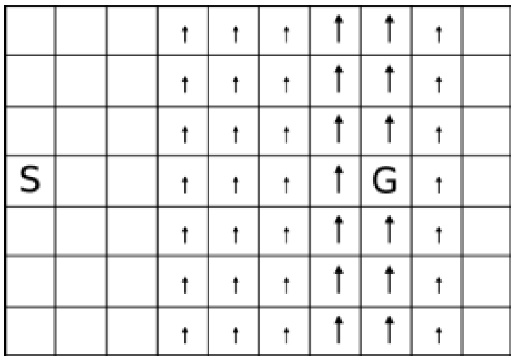

Figure 1: Stochastic Windy Gridworld: 10x7 grid. 'S' is the agent's starting point and 'G' is the goal. The small arrows in columns 3,4,5,8 indicate that the wind moves the agent upwards by one step, whereas the big ones in columns 6,7 indicate a two step movement.

3 Dynamic Programming

Dynamic programming is a method between planning the moves of the agent in an environment and reinforcement learning. When dealing with this, it is a requirement to assume that the environment is a finite Markov Decision Process, which means that the next state depends only on the current state and the actions available in it. It is a model based approach since it is necessary to have full access to the environment. It is based on the tabular method which consists in forming a table Q(s,a) containing the values on each state for each action that the agent can perform. The sets that contain the states and actions are S and A, respectively. Furthermore, the probabilities that the agent from a state s based on the action a can possess to state s' and future rewards based on the transition are known. Despite the fact that Dynamic Programming guarantees to find a solution, it is not a preferred method due to two limitations: it is an appropriate method only in the case of small dimensions, since the construction of the table requires a large memory capacity and moreover in most cases that Reinforcement Learning tries to deal with, the model is rarely known. The following subsections, Methods and Results are dedicated to the explanation of the implementation of the algorithms and the interpretation of the results obtained, respectively.

3.1 Method

3.1.1 Policy

The policy used in Dynamic Programming is deterministic and it is used to determine the action the agent is going to take in the state s: $\pi(s)$. In our case the action is selected based on the function:

$$\pi(s) = argmax_a Q(s, a).$$

This function selects the index of the action that leads to the maximum state-action value from the table Q (s, a). In addition, in case two or more state-action values are the same, it chooses one at random.

3.1.2 Update

The Dynamic programming method is based on the Q-value algorithm in which for every state to another based on the most preferable action, the estimates of the state-action values are updated following the equation:

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

where Q(s,a) is a table with all the state-action values, $p(s'|s, a)$ are the probabilities that action a in state s will transition to state s', r(s,a,s') is the reward received after action transitions state s to state s', γ is the discount factor representing the difference between future and present rewards and last but not least, the $max_{a'} Q(s', a')$ corresponds to the maximum state-action value of the value of the state s' and the action a'.

3.1.3 Maxixum error

To terminate the algorithm, a variable named max_error is initialized to zero. Specifically, the program continues to run until we get a max error less than a threshold of 0.001. The maximum error is updated at each iteration following this formula:

$$max\_error = max(max\_error, \left|Q(s, a) - \hat{Q}(s, a)\right|),$$

which consists in finding the maximum value between the maximum error calculated in the previous iteration and the absolute value of the difference between the previous Q (s, a) and the updated Q(s',a') for every state a and action a.

3.2 Results

The Dynamic Programming algorithm reaches the final state in 18 iterations. In the following figures (Fig.2, Fig. 3) we can observe the progression of Q-value iteration during execution, following the algorithm implemented, and the estimates at each state-action, specifically at the midway (9th iteration) and at convergence (last step). The figure of the progression of Q-value iteration during execution at the beginning is omitted for simplicity since the estimates are all zero. Moving on to the latest iterations we expect the estimates surrounding the target to be higher to spur the agent to move nearby and then reach the target. Obviously in this case the distant cells have a lower value to prevent the agent from going back. This phenomenon is clearly evident in the figures as we expected. It is important to note that the values in the final state 'G' are zeroes, thus underlining the fact that the goal (s = 52) has been reached and that the agent no longer has to move to another cell. This is because the environment has been constructed in such a way that if s is terminal state it is assumed that the probability of transitioning to another step is 1 since the objective has already been reached and the rewards are set to zero. Therefore, this zero reward self loop explains the fact that the algorithm continues to converge. Also a very important thing to note is that, the converged optimal value at the start V * (s = 3) under the optimal policy which is equivalent to the maximum estimate that occurs at the initial state is approximately 18.31. It indicates the absolute value of the rewards gained to reach the final state, since the reward for each action is -1.

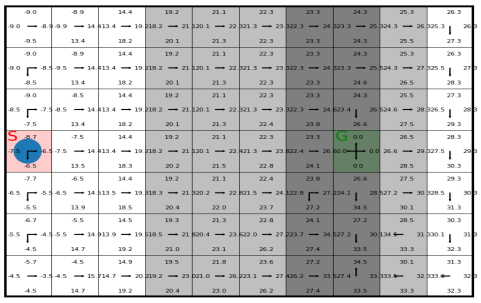

Figure 2: Progression of Q-value iteration during execution: 9th step

Last but not least, as we expected at each iteration the

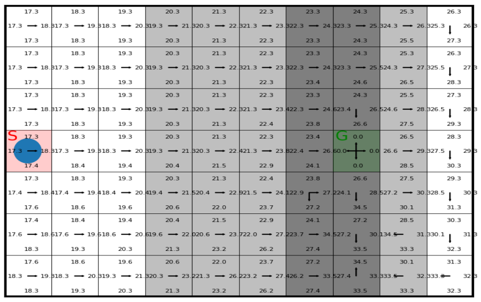

Figure 3: Progression of Q-value iteration during execution: 18th step

max error decreases. At iteration 1 the error has a value of 29.704, whereas at iteration 18, when the goal is reached by the agent, the max error is 0.00032, which is obviously less than the threshold set.

3.3 Average reward per timestep

The average reward per timestep under the optimal policy is computed by taking into consideration three components:

- V*(s = 3): the optimal value at the start state (18.31)
- The magnitude of the final reward (+35)
- The magnitude of the reward on every other step (-1)

  Considering x is the number of steps, the start state equals to:

$$V*(s=3) = (x-1) \times (-1) + 35$$

  . Therefore, the average number of steps is: x= 35- V * (s = 3) +1 = 17.7, which concludes, that the average reward per timestep is: $\frac{V*(s=3)}{x} = 1.03$

Exploration

This section is dedicated to implement a model free reinforcement learning algorithm. In this case, as opposed to Dynamic Programming, we have no longer access to the model and the transition probabilities are unknown. Since we lack of crucial information, the role of the transition function is replaced by an iterative sequence of environment samples. Specifically, we are going to implement the off-policy Q-learning algorithm.

3.4 Methods

In this algorithm the value estimates are bootstrapped directly after one transition (1-step). However, in the following sections we are going to discuss about depth and n-step methods (see section Back-up: Depth of target) where bootstrapping is applied later.

3.4.1 Policy

Since during the exploration, is not guaranteed that the agent will move through all the states using only the greedy policy, it is necessary to explore more and try something novel. In this case we compare 2 policies:

- E-greedy:

$$\pi(a|s) = \begin{cases} 1.0-\epsilon, & if \quad a = \underset{b\in A}{argmax}\hat{Q}(s,b) \\ \epsilon/(|A|), & otherwise \end{cases} \tag{1}$$

  where $\pi(a|s)$ is the probability of taking action a in state s under stochastic policy π, ϵ is the probability of taking a random action in a greedy policy, b are the actions taken from set A, and argmax has been introduced before.

  In words, we generate uniformly a random number n ∈ [0,1], and if the ϵ set is less than n we follow the greedy policy used in Dynamic Programming where we select the action that returns the maximum state-action estimate, otherwise the action in each state is selected randomly.

- Boltzmann policy:

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

  where $\tau \in (0,\infty)$ is a temperature parameter. The Boltzmann policy function is based on the softmax function, it returns a probability and gives a higher probability to actions with a higher current value estimate. The action is then selected randomly based on the probabilities given. For $\tau \to 0$ the policy becomes greedy whereas for $\tau \to \infty$ the policy becomes uniform/random because the probability of each action is the same (1/len(A)), hence, for each state the action is uniformly randomly selected.

3.4.2 Update

The update step is based on bootstrapping which is a process used to adjust old values with new updates to find an approximate final value. We therefore, speak of temporal difference learning (TD) where the differences in values between two time steps are practically used to calculate the value in the next step. Since we are dealing with a model-free algorithm, each time an action is executed, the environment provides new data for the next iteration in terms of the reward observed and the next state. By using the new observed data (s_t, a_t, r_t, s_{t+1}) the goal is to compute a new estimate for the state-action value at the timestep t. Q-learning selects the best action in the state s_t by looking at the reward for the current state-action combination, and the maximum rewards for the state s_{t+1}.

The update function can be summarized in two steps:

1. Back-up target G_t:

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

   where the r_t is the reward obtained at the timestep t, γ is the discount rate as mentioned above, and $max_{a'} \hat{Q}(s_{t+1}, a')$is the action that returns the maximum estimate at step t+1.

2. Tabular learning update:

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

   where $\alpha \in (0, 1]$ denotes the learning rate, G_t is the target function and $\hat{Q}(s, a)$ is the table with state and actions that has to be updated based on previous data.

It is sufficient to observe the update formula to be able to deduce that the state action values are updated by backing up values of another action without considering the action selected by the policy. In other words, the Q-values are updated using the state-action values of the next state s_{t+1} and the greedy action, and not by using the policy function. Therefore, the phenomenon where all available information is collected and used to construct the best target policy is called off-policy, which confirms the fact of which we are already aware of, that Q-learning is an off-policy algorithm (Plaat, 2022).

It is worth highlighting the importance of selecting the optimal value of the learning rate, since a learning rate that is equal to zero leads to a slow but stable learning, while on the other hand, a learning rate that is equal to 1 leads to a fast but less stable learning. The impact of this parameter is going to be further explained in the Results section where plots with different settings are reported.

3.5 Results

Multiple experiments have been carried out in order to test several parameters Specifically, the experiments have been repeated for a number of 50 times ($n_{repetitions} = 50$) in 50000 timesteps ($n_{timesteps} = 50000$). The figure below represent the plots based on the average of the results obtained through the 50 repetitions.

The parameters tested in the experiments are: γ = 1.0 , three different ϵ for the ϵ-greedy policy ϵ = [0.01, 0.05, 0.2] and three different settings for the Boltzmann policy τ = [0.01, 0.1, 1.0]. The learning rate has been set to α = 0.25.

Looking at the curves in Fig. 4 we can see an approximately similar behavior for any parameter experimented. Specifically, the average reward that the algorithm reaches is between 0.5 and 0.6. Moreover, it roughly needs the same number of timesteps to converge ($n_{timesteps} \approx$ 50000) and its learning is fast. The only differences that can be noticed concern the curves of ϵ-greedy with ϵ = 0.2, which has a fast learning at the beginning however it grows more slowly until it reaches an average of reward on 0.35 and of the softmax with τ = 1.0, which has slower learning and needs approximately twice as many timesteps to converge ($n_{timesteps} \approx$ 10000).

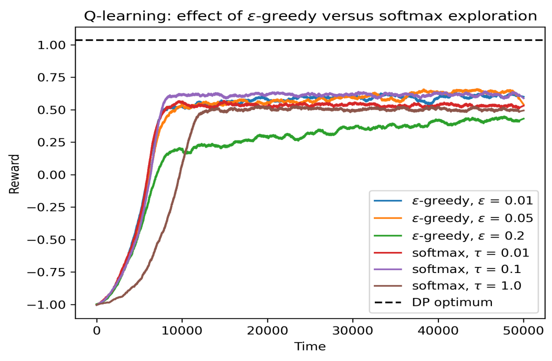

Figure 4: Q-learning plot: difference between ϵ-greedy and softmax policy.The x-axis represents the timestep represented and the y-axis the average reward through all 50 repetitions. The parameters tested for the ϵ-greedy policy are: ϵ = [0.01, 0.05, 0.2], while for the Boltzmann policy τ = [0.01, 0.1, 1.0]. The learning rate has been set to α = 0.25 and γ = 1.0. The DP optimum refers to the average reward per timestep obtained in Dynamic Programming. The plot suggests that the algorithm converges with any parameter. No huge difference can be discerned between them. The rewards obtained are approximately between 0.4 and 0.6 with the exception of the ϵ-greedy curve with ϵ = 0.2 which reaches more or less 0.3. All of them have stable learning, and more or less converge at the same time, except softmax with τ = 1.0 which needs more timesteps.

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

The goal of this section is to implement the SARSA algorithm to compare the on-policy method versus the off-policy method implemented in the previous section (Q-learning). The difference between these two is the way the information is backed-up.

3.6 Methods

The following SARSA algorithm is a 1-step method as Q-learning in order to analyze the difference between them.

3.6.1 Policy

The policy function implemented is the same as the Q-learning one, including both ϵ-greedy and Boltzmann policies.

3.7 Update

In this case, the update function is based on the following observations $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. Unlike the Q-learning algorithm, SARSA takes into account and backs up one more observation: the next action a_{t+1}, and updates it state-action values using the Q-value of the next state and the current policy's action. The SARSA update function can be summarized in two steps:

1. Back-up target:

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

   where the r_t is the reward obtained at the timestep t, γ is the discount rate as mentioned above, and $\hat{Q}(s_{t+1}, a_{t+1})$ is the state-action value in the next state (s_{t+1}) and next action (a_{t+1}) .

2.

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

   where $\alpha \in (0, 1]$ denotes the learning rate, G_t is the target function and $\hat{Q}(s, a)$ is the table with state and actions that has to be updated based on previous data.

As stated before, the only thing that differs the SARSA algorithm from Q-learning is the value they bootstrap at the next state. The tabular learning update function remains the same. This time, we are dealing with an on-policy learning method, hence, SARSA backs up the value of the action we actually take by learning the value function of the policy executed. In other words, the agent moves on to better actions by taking into consideration the function policy (Plaat, 2022).

3.8 Results

Again, the experiments have been repeated for a number of 50 times ($n_{repetitions} = 50$) in 50000 timesteps. ($n_{timesteps} = 50000$). The figures below represent the plots based on the average of the results obtained through the 50 repetitions.

The parameters tested in the experiments are: γ = 1.0 , three different learning rates α= [0.05,0.2,0.4], ϵ= 0.05 for the ϵ-greedy policy and τ= 1.0 for the Boltzmann policy. By observing Fig.5 we can demonstrate what we were already aware of regarding the learning rate: the more the learning rate increases, the faster the algorithm learns. Specifically, with a learning rate of 0.05, both Q-learning and SARSA (with an ϵ-policy) have a much slower but stable trend, they begin to converge after 40,000 timesteps but they eventually reach an average reward of 0.5. With a learning rate equal to 0.2, the SARSA algorithm reaches an average reward of 0.6 and performs slightly better than Q-learning. While with a learning rate equal to 0.4 Q-learning learns quickly until a certain timestep, after the timestep its learning continues to increase slightly, reaching better results than SARSA which instead has a behavior with more fluctuations. It is important to note that the latter achieves the same result as SARSA with a learning rate equal to 0.02. Finally, another important observation is that both Q-learning and SARSA methods don't converge within the number of timesteps set if the softmax policy is utilized and a low learning rate (α= 0.05) is set. In general we can conclude that both algorithms, despite the difference between the back up method and the policy used, behave almost equally.

Back-up: Depth of target

This section deals with the previously mentioned topic concerning the depth of the back up. Up to now the implemented algorithms were based on the 1-step method, where bootstrap of the estimated value is applied after a transition. In this case, instead, we try to implement the n-step methods where multiple rewards are added before applying the bootstrap. Specifically, we will implement n-step Q-learning. Furthermore, a second n-step method will also be applied, by implementing the Monte Carlo update, where rewards are added directly to the end of the episode.

3.9 n-step Q-learning

The n-step Q-learning method bootstraps in the same way as a 1-step Q-learning algorithm, however, the depth of the back up is different. This time, we don't bootstrap directly after a transition, but conversely, we sum up multiple rewards before updating.

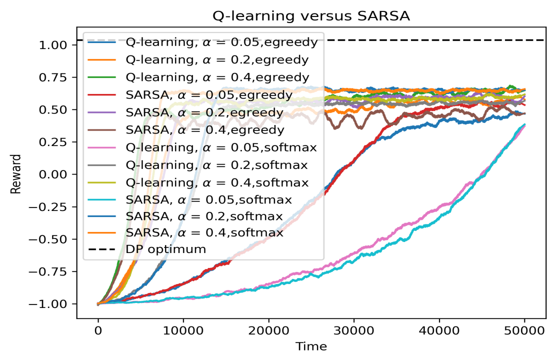

Figure 5: Q-learning vs SARSA policy.The x-axis represents the timestep represented and the y-axis the average reward through all 50 repetitions. The parameters tested are: $\epsilon = 0.05$, τ= 1.0, three learning rates α = [0.05,0.2,0.4] and $\gamma = 1.0$. The DP optimum refers to the average reward per timestep obtained in Dynamic Programming. The plot suggests that the algorithms converge with any parameter for both algorithms. Curves with a lower learning rate converge slower but are more stable, whereas curves with higher learning rate converge faster but are less stable. The combination of softmax policy and a low learning rate yields to curve that does not converge.

3.9.1 Methods

Policy

The policy function implemented is the same as the Q-learning and SARSA algorithms, including both ϵ-greedy and Boltzmann policies.

Update

The update function can be again summarized in two steps:

1. Back-up target:

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

where n is the target depth, γ is the discount parameter, r_{t+i} is the reward at timestep t + i depth, and $max_a Q(s_{t+n}, a)$ is the action that gives the maximum Q-value at the state s_{t+n}.

2. $$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)]$$

which is the tabular update seen at the previous algorithms.

The n-step Q-learning is not exactly off-policy method, because the first n rewards are generated from the current policy and hence the target principally follows the behavioral policy.

3.10 Monte Carlo

Other than the n-step Q-learning, another way to introduce a depth is by implementing the Monte Carlo update. In this subsection we are going to report the formulas used for the implementation of the Monte Carlo algorithm which differs from the n-step Q-learning as it omits bootstrapping and simply sums all the rewards up to the end of the episode.

3.10.1 Methods

Policy

The policy function implemented is the same as the Q-learning and SARSA algorithms, including both ϵ-greedy and Boltzmann policies.

Update

The update function can be again summarized in two steps:

1. Back-up target:

$$G_t = \sum_{i=0}^{\infty} (\gamma)^i \cdot r_{t+i}$$

where γ is the discount parameter, r_{t+i} is the reward at timestep t + i depth

2. $$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t)]$$

which is the tabular update seen at the previous algorithms.

3.11 Results

Again, the experiments have been repeated for a number of 50 times ($n_{repetitions} = 50$) in 50000 timesteps. ($n_{timesteps} = 50000$). The figures below represent the plots based on the average of the results obtained through the 50 repetitions.

The parameters tested in the experiments are: $\gamma = 1.0$, α= 0.25, ϵ= 0.05 for the ϵ-greedy policy and τ= 1.0 for the Boltzmann policy. In addition, different depths of the target have been tested for the n-step Q-learning and Monte Carlo, with n = [1,3,5,10,20,100].

By observing Fig. 6, we can conclude that the algorithm that performs best is the 1-step Q-learning with a greedy policy. On the contrary, we can notice that the softmax policy whenever the step depth is large, such as for n = 20,100, has no improvement and does not find a solution. The same thing also applies to Monte Carlo no matter what policy is used. Finally, the algorithm under the other parameters set, has a mediocre performance and the curves seem to fluctuate a lot, but, with the right parameter tuning, the n-step can be very promising and achieve better results.

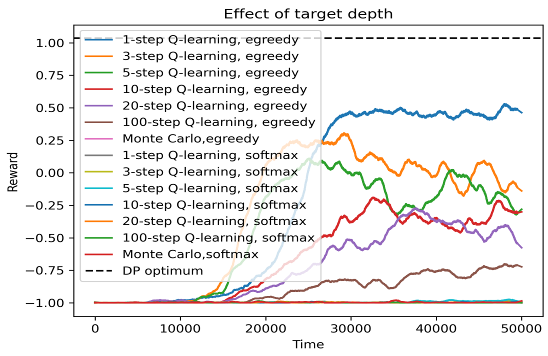

Figure 6: Effect of target depth.The x-axis represents the timestep represented and the y-axis the average reward through all 50 repetitions. The parameters tested are: ϵ = 0.05, τ= 1.0, learning rates α = 0.25, γ = 1.0 and several depths n = [1,3,5,10,20,100].The DP optimum refers to the average reward per timestep obtained in Dynamic Programming. The best curve belongs to the 1-step Q-learning which reaches an average reward of 0.5. The softmax policy for deep n-steps (n = 20,100) as well as Monte Carlo with both ϵ-greedy and softmax policies do not perform well.

Reflection and Conclusions

Generally, dynamic programming is not considered a method that entirely belongs to reinforcement learning, on the contrary, it is a method that connects the world of planning and reinforcement learning. This is because in most cases we have to deal with model-free methods, which is not possible when it comes to Dynamic Programming. An advantage of the latter is that it guarantees to find a solution by finding the optimal policy, and it always converges. However, since it's a tabular method (like the other algorithms that were considered in this report), in case we are dealing with high dimensions (such as testing on a larger environment) the so called 'curse of dimensionality' applies, and therefore we need a huge memory capacity to store all possible discrete states. In this case, thanks to Machine Learning methods, we can deal with such issues, since numerous dimensionality reduction techniques have been implemented. As far as exploration is concerned, no major differences were identified between softmax and ϵ-greedy policies in the trend of the curves. Regarding the comparison between on and of policy methods, the major difference between on and off policy is based on how the back up takes place, which has already been addressed in the previous sections. In addition, The strengths of on-policy methods are that they converge quickly and theirs convergence is more stable (low variance). This is due to the fact that on-policy targets follow the behavioral policy. However, the target policy is updated with sub-optimal explorative rewards which leads to dealing with not optimal sample efficiency. On the others hand, off policy methods usually in their convergence some fluctuations are present, hence, is less stable (low bias). As stated earlier, the n-step Q-learning is not exactly a off-policy method, since the first n rewards are sampled from the current policy, hence, the target principally follows the behavioral policy. (Plaat, 2022) As far as the target depth is concerned, for this task the n-step Q-learning with the right parameter optimization (e.g. learning rate and not deep target ($n < 10$)) seems to be very promising and can easily "beat" the 1-step Q-learning, however the 1-step Q-learning method is more likely to converge on the optimal policy since we are dealing with a simple task and the greedy approach is preferred. Monte Carlo is a method, which contrary to the n-step, does not propagate information very quickly since it omits bootstrapping and one must wait until the end of an episode to update. We can also see this fact in Fig. 6 where it is evident that n-step method learns faster.

Finally, it would be interesting to apply a Grid Search to get the most promising parameters and see how things change when we implement these algorithms in a new environment.

References

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

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