Assignment 1
Reinforcement Learning 2023

Guillem Casals Ruesga [*1]

Abstract

In this assignment, we will study a range of basic principles in tabular, value-based reinforcement learning. In particular, we will study the following topics of Dynamic Programming (DP) and Model-Free RL. In the latter we will explore the topics of exploration and back-up, comparing off-policy versus on-policy methods as well as the depth.

1. Introduction

As stated, this assignment is to help study a range of basic principles of RL. To do so, we will divide it in four parts: Dynamic Programming, Exploration, Back-up: On-policy versus off-policy target and Back-up: Depth of target. Each one of these topics has its own methods, experiments, and results. All of them use the same environment (same problem) of the Stochastic version of Windy Gridworld, and hence we will be able to see the differences between each approach when trying to achieve the same solution.

2. Environment

The environment decided for this assignment is the Stochastic version of Wind Gridworld, which is based on (Sutton & Barto, 2018), in the Example 6.5 at page 130. However, compared to the book version, in this case the vertical wind is now stochastic and only blows 80% of times.

The environment consists of a 10x7 grid, where at each cell we can move up, down, left or right. We start at location $(0, 3)$ (indexing starts at 0), indicated in the Figure 1 by 'S'. Our goal is to move to location $(7, 3)$, indicated by 'G'. However, as mentioned earlier, a special feature of the environment is that there is a 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. The wind does not always blow, but is randomly present on 80% of the occasions (which makes the environment stochastic). The reward of the agent at each step is -1, while reaching the goal gives a reward of $+40$, and terminates the episode.

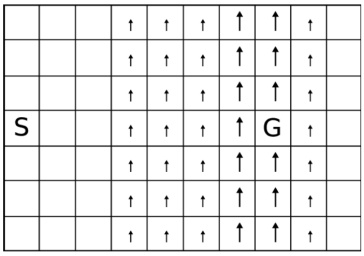

Figure 1. Image representation of the environment that every approach will try to solve.

3. Dynamic Programming

Dynamic programming is a bridging method between planning and reinforcement learning. It assumes full access to a model of the environment, i.e., we can get $p(s'|s, a)$ and $r(s, a, s')$ for any state s and action a. DP is guaranteed to find the optimal solution, but it requires a model (which is not always available) and suffers from the curse of dimensionality as all tabular methods actually do. In particular, we will study the Q-value iteration algorithm. This algorithm sweeps through all state-action pairs, each time updating the estimate of a state-action value based on the following equation:

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

3.1. Methods

We implemented this approach by completing the class `QValueIretationAgent()` and the function `Q_value_iteration()` in the file `DynamicProgramming.py`.

[*]Equal contribution [1]Master on Computer Science: Artificial Intelligence, University of Leiden, Leiden, Netherlands. Correspondence to: Guillem Casals Ruesga <s3741486@vuw.leidenuniv.nl>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

First, we implement the `select_action()` function, which selects the greedy best action a of all the possibles in the state s:

$$\pi(s) = argmax_a[Q(s,a)] \tag{2}$$

Secondly, the `update()` function is also implemented. As the name indicates, it is responsible for updating the values of $Q(s,a)$ using the Q-iteration update sown in Eq. 1, hence updating in-place the $Q(s,a)$ from the policy $\pi(s)$.

Finally, the function `Q_value_iteration()` is completed by executing the Q-value iteration making use of the update function previously implemented.

Algorithm 1 Q-value iteration (Dynamic Programing)

Input: Threshold $\eta \in \mathbb{R}^+$, Transition probabilities $p \leftarrow p(s'|s,a)$, Rewards $r \leftarrow r(s',a,s)$
Result: The optimal policy $\pi^*(s)$.
Initialization: A state-action value table $\hat{Q}(s,a) = 0 \; \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
$\triangle \leftarrow \infty$
while $\triangle < \eta$ **do**
  for each $s \in \mathcal{S}$ **do**
    for each $a \in \mathcal{A}$ **do**
      $x \leftarrow Q(s,a)$
      $\hat{Q}(s,a) \leftarrow update(s,a,p,r)$
      $\triangle \leftarrow \max(\triangle, |x - \hat{Q}(s,a)|)$
    end for
  end for
end while
$\pi^*(s) = argmax_a(\hat{Q}(s,a))$
Return: $\pi^*(s)$

3.2. Results

In this part we executed the program with the aim to compute the average reward per timestep under the optimal policy, and to evaluate the Q-value iteration during the execution. To achieve this, we executed the experiment once and computed the mean reward per timestep under the given policy 1.3. Since there is a randomness attached to it, we decided to run repetitions of the experiment and average our results over repetitions. This gave us the average reward per timestep: 1.329, that we will use in the next experiments to set the optimum.

The progression of the Q-value iteration during the execution until convergence is reached is shown in the following Figures 2, 3 and 4.

The values observed in each cell are the $Q(s,a)$ values for all possible actions in each state. Since we used the selection based on the greedy option, the arrow (that indicates the best possible action to take in each state) will always point to the higher number.

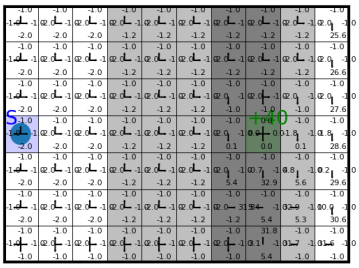

Figure 2. Q-value iteration during execution, estimates of each state-action at the beginning.

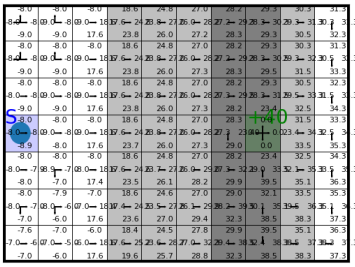

Figure 3. Q-value iteration during execution, estimates of each state-action at midway.

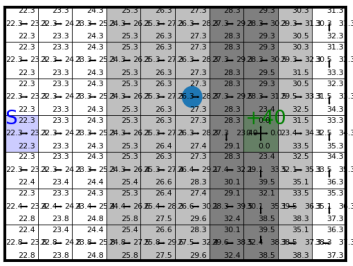

Figure 4. Q-value iteration during execution, estimates of each state-action when convergence is reached.

For example, if we compute the converged optimal value at the start $V^*(s=3)$, where $V^\pi(s)$ is defined as the return we expect to achieve when an agent starts in state s and follows policy π, as:

$$V^\pi(s) = E_{\tau_t \; p(\tau_t)}[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s] \tag{3}$$

The result we obtain is $V^*(s=3) = 23$ which means that the expected cumulative reward that the optimal policy achieves is 23 since we chose the greedy option, therefore we will move to the right.

These values are possible to know for every cell before we start to move, because in the DP we have full access to the environment. Once convergence is reached, meaning the error is lower than the given threshold (as shown in the presented Algorithm 1), all the states (cells) have already selected the best action to take in order to minimize the error, therefore minimizing the amount of steps needed to reach the goal, to consequently maximize the final reward.

TERMINAL PROBLEM DO NOT KNOW, ASK

4. Exploration

In this section we switched to the (model-free) reinforcement learning setting. Therefore, in this case, we did not have access to the model, as it would happen in the real world, where executing an action permanently brings you to the next state. Since we could not sweep through all states anymore, we proceeded in episodes from the start state. This meant that it is possible that not all states are visited under a greedy policy. For that, we needed to introduce exploration into the action selection.

For this, we compared two types of policies:

- **The ϵ-greedy policy**:

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon\frac{|\mathcal{A}|-1}{|\mathcal{A}|}, & \text{if } a = argmax_{b\in\mathcal{A}}\hat{Q}(s,b), \\ \epsilon/|\mathcal{A}|, & \text{otherwise} \end{cases} \tag{4}$$

  With a small probability ϵ we selected a random action, ensuring exploration, otherwise the greedy action is taken. The parameter ϵ controls the amount of exploration ($\epsilon = 0$ gives a greedy policy, $\epsilon = 1$ gives a uniform/random policy).

- **The Boltzmann policy**:

$$\pi(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{b\in\mathcal{A}} e^{\hat{Q}(s,b)/\tau}} \tag{5}$$

  Where $\tau \in (0, \infty)$ denotes a temperature parameter. This approach gives a higher probability to actions with a higher current value estimate, but still ensures exploration of other actions than the greedy one. The temperature τ controls the amount of exploration: for $\tau \to \infty$ the policy becomes uniform/random, and for $\tau \to 0$ the policy becomes greedy.

The second crucial step of an RL algorithm is the update. After executing an action, the environment gives new data, the observed reward, and next state. Therefore, after timestep t, the data received is $< s_t, a_t, r_t, s_{t+1} >$. The task at hand was to compare different ways to use this data to compute a new estimate for the state-action value at s_t, a_t.

We used the 1-step Q-learning update. First we computed the new back-up estimate/target G_t as in Eq. 6 and then apply the tabular learning update Eq. 7

$$G_t = r_t + \gamma max_a \hat{Q}(s_{t+1}, a') \tag{6}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)] \tag{7}$$

Where $\alpha \in (0, 1]$ denotes the learning rate.

4.1. Methods

The first thing was to implement the `select_action*(` function. We implemented the two policies mentioned earlier where the selection of the policy is determined by a string that is either $'egreedy'$ or $'softmax'$.

The `update()` function was done by applying the Eq. 6 and 7, as stated before.

Finally, the implementation of the Q-learning algorithm has been made as shown in Algorithm 2.

Algorithm 2 Tabular Q-learning

Input: Learning rate $\alpha \in (0, 1]$, Discount parameter $\gamma \in [0, 1]$
Initialization: A state-action value table $\hat{Q}(s, a) = 0$ $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
$s \sim p_0(s)$; `/* Initialize state */`
while *budget* **do**
  $a \sim \pi(a|s)$; `/* Select action */`
  $s_{next}, r\ p(r, s_{next}|s, a)$; `/* step */`
  $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)]$; `/* update */`
  if s_{next} *is terminal* **then**
    $s \sim p_0(s)$; `/* Reset environment */`
  else
    $s \leftarrow s_{next}$
  end if
end while
Return: $\hat{Q}(s, a)$

4.2. Experiments

We performed a systematic experiment, comparing the ϵ-greedy and the Boltzmann policies with different settings for the exploration parameters (ϵ and τ respectively). Concretely, we tried the ϵ-greedy with $\epsilon = [0.02, 0.1, 0.3]$ and softmax with $\tau = [0.01, 0.1, 1.0]$. Thus, this allowed us to see how the different approaches work in terms of exploration and exploitation.

4.3. Results

The results of the experiment are shown in the following Figure 5.

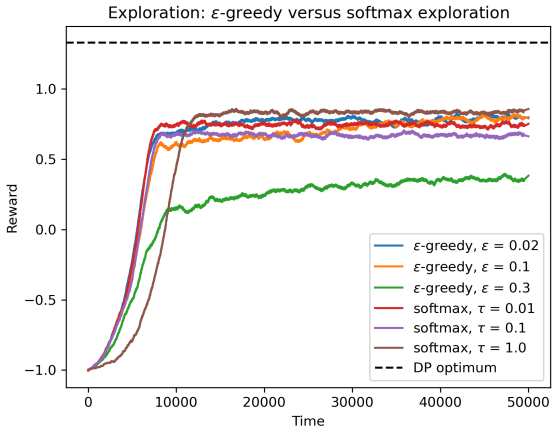

Figure 5. Both exploration methods for different values of ϵ and τ

From this figure, we can clearly see that all methods have a similar performance, except when we use a large value for ϵ. Having a large value for ϵ from the ϵ-greedy policy determines a much more exploration-wise policy, therefore if the value is too high it would be like having a completely uniform/random policy. It therefore makes sense that giving a large exploration rate than needed to the ϵ-greedy policy would diminish its performance.

All in all, we obtain good and similar results with all the tested values. However, it is far from the optimum set by the Dynamic Programming (3.2), which makes total sense since we do not have access to the environment beforehand given that we are a model-free reinforcement learning setting. Not having the information beforehand means that a large number of tries have to be done just to search for a policy (path) that reaches the solution, and much more to find the optimal one. We might not even find it with the proposed budget. That is why it does not achieve the optimal performance found.

5. Back-up: On-policy versus of-policy target

The second important part of any RL algorithm is the way we back-up information. A major distinction is between off-policy back-ups (like Q-learning) and on-policy back-ups (like SARSA). This part is focused on the one-step case. The back-up equation for Q-learning was already implemented in Section 4 in En. 6 and 7.

And, on the other hand, the back-up equation for SARSA, given the observations $< s_t, a_t, r_t, s_{t+1}, a_{t+1} >$ is:

$$G_t = r_t + \gamma max_a \hat{Q}(s_{t+1}, a_{t+1}) \tag{8}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)] \tag{9}$$

The major difference between these two is the value they bootstrap at the next state. Q- learning plugs in the value of the *best possible action at the next state*, and thereby attempts to learn the value of the optimal policy. While, SARSA backs up the value of the action we actually take (which may be exploratory and not the one with the currently optimal estimate). SARSA therefore learns the value function of the policy we actually execute (which includes exploration). Both approaches have their benefits and problems, and in this section an experiment was carried away in order to compare them.

5.1. Methods

Since the Q-learning approach has already been implemented, in this section we focused only on the SARSA. To implement the action selection we used the same as in the Q-learning approach stated in Section 4.1. And for the update, as in the Q-learning, the implementation of the two equations Eq. 8 and 9 was performed.

Last but not least, the implementation of the SARSA algorithm emulates the Algorithm 3, that results very similar to the Q-learning one, but changing the update employing the new aforementioned observations $< s_t, a_t, r_t, s_{t+1}, a_{t+1} >$.

5.2. Experiments

We performed a more thorough comparison between Q-learning and SARSA by changing the learning rate value $\alpha = [0.02, 0.1, 0.4]$ in both methods with 50 repetitions. This allowed us to see the difference between the two methods when the value of the bootstrap changes accordingly to the hyperparameter.

5.3. Results

As mentioned above, in Section 5.2. We changed the learning rate on both methods and evaluated through all repetitions.

Looking at Figure 6 we can denote a good performance on both approaches, although one thing that catches the sight is how both suffer a great loss when the learning rate is very small. This makes total sense, since advancing that slowly would take a lot of repetitions to arrive at the same point where the other experiments are. Note that the best performing Q-learning is when α has the highest value, whilst SARSA reaches that when $\alpha = 0.1$. That is a

Algorithm 3 Tabular SARSA

Input: Learning rate $\alpha \in (0, 1]$, Discount parameter $\gamma \in [0, 1]$
Initialization: A state-action value table $\hat{Q}(s, a) = 0$ $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
$s \sim p_0(s)$; /* Initialize state */
$a \sim \pi(a|s)$; /* Select action */
while *budget* **do**
  $s_{next}, r\ p(r, s_{next}|s, a)$; /* step */
  $a_{next} \sim \pi(a_{next}|s_{next})$
  $G_t \leftarrow r_t + \gamma max_a \hat{Q}(s_{next}, a_{next})$
  $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)]$; /* update */
  if s_{next} *is terminal* **then**
    $s \sim p_0(s)$; /* Reset environment */
    $a \sim \pi(a|s)$
  else
    $s \leftarrow s_{next}$
    $a \leftarrow a_{next}$
  end if
end while
Return: $\hat{Q}(s, a)$

consequence of the SARSA being a method which explores more in nature, so a smaller learning rate makes it steadier that a larger one, on the other hand Q-learning always goes for the greedy option, which makes it perform better with a bigger α.

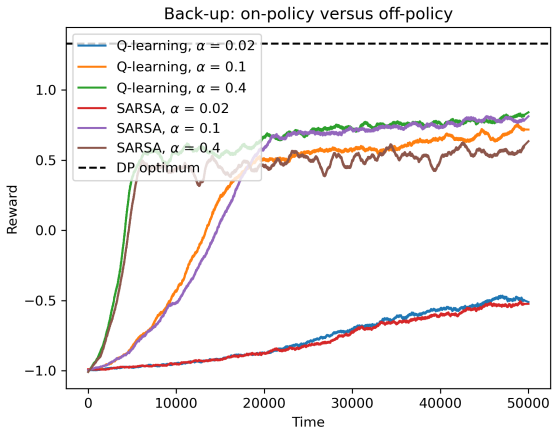

Figure 6. Q-learning and SARSA for different values of α

In our environment the Q-learning and SARSA methods seem efficient due to the simplicity of the problem. Therefore, makes it difficult to choose one before the other. Nevertheless, we think that the Q-learning method is more suited to this problem since the basic setting makes it more efficient to just look for the best possible action in each state (greedy policy). But in another environment with a more complex settings, and a clear need for exploration the SARSA approach can result better than the Q-learning, where the latter can get stuck in a local maxima (suppose we try to maximize the reward) versus the SARSA that wold explore more and could find the optimal or a better one.

6. Back-up: Depth of target

Depth of target is the other important aspect of the back-up. So far, we have only looked at 1-step method in Section 5, which directly bootstrap a value estimate after one transition. When applying depth of target, we can also sum multiple rewards in a trace before we bootstrap, which leads to n-step methods (n-step Q-learning or n-step SARSA, depending on the way you bootstrap). In this section we used the n-step Q-learning, which computes the following target.

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

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)] \tag{11}$$

Note that, although it is called n-step Q-learning (due to the maximization over the last action in Eq. 11), it is not a full off-policy method, since the first n reward are of course sampled from the current policy (and the target therefore mostly follows our behavioral policy). Another approach to the N-step method could be to omit bootstrapping altogether. By simply sum all rewards up to the end of the episode (or up to some maximum timestep after which we terminate the episode). The is know as the Monte Carlo update,that goes as follows:

$$G_t = \sum_{i=0}^{\infty} (\gamma)^i r_{t+i} \tag{12}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[G_t - \hat{Q}(s_t, a_t)] \tag{13}$$

This section aims to compare this two different approaches on the N-step methods to work with depth.

6.1. Methods

To implement the two approaches mentioned above. First, the same selection mechanism as in the previous algorithms that is mention in Section 4 and4.1. In this case, the update function was more complex than in the other sections, se we will also introduce a small pseucode of how we implemented that in each of the two approaches.

In N-step Q-learning, using the Eq. 10 and 11 as follows:

Algorithm 4 N-step Q-learning update function

Input: Data received in vector form $states, actions, rewards$ and target depth n
$T_{ep} \leftarrow len(actions)$; /* Episode length */
for $t = (0, ..., T_{ep} - 1$ **do**
  $m \leftarrow min(n, (T_{ep} - t))$; /* Rewards left to sum */
  $G_t \leftarrow 0$
  for $i = 0, ..., m - 1$ **do**
    $G_t \leftarrow G_t + (\gamma)^i rewards_i$
  end for
  if $states_{t+m}$ is $terminal$ **and** $t + m < T_{ep}$ **then**
    $G_t \leftarrow G_t + (\gamma)^m max_a \hat{Q}(states_{t+m}, actions_t)$
  end if
  $\hat{Q}(states_t, actions_t) \leftarrow \hat{Q}(states_t, actions_t) + \alpha[G_t - \hat{Q}(states_t, actions_t)]$
end for

The Monte Carlo update function is rather easily in comparison, since we only have to sum all the rewards until the end of the episode:

Algorithm 5 Monte Carlo update function

Input: Data received in vector form $states, actions, rewards$
$T_{ep} \leftarrow len(actions)$; /* Episode length */
$G_t \leftarrow 0$
for $i = T_{ep} - 1, ..., 0$ **do**
  $G_{t_i} \leftarrow (\gamma)^{i+1} + rewards_i$
  $\hat{Q}(states_i, actions_i) \leftarrow \hat{Q}(states_i, actions_i) + \alpha[G_{t_i} - \hat{Q}(states_i, actions_i)]$
end for

With all that, we could finally implement the algorithms themselves. We used the pseudo-code in the instructions as help, but since we divided the taks of the algorithm presented in the instructions between the update and our final algorithm it resulted somewhat different. Likewise, the Monte Carlo approach has a different structure than the presented into the instructions. But, a similar one to to N-step Q-learning, since the main change lies in the update function explained earlier in this section.

We implemented a general algorithm for both of them, since there was no downfall due to the similarities shared in both. This algorithm ended up with the structure presented as follows in Algorithm 6.

6.2. Experiments

Comparing different back-up depths, needed a more systematic experiment: For that, we tried different back-up depths: $n = [1, 3, 10, 30]$ in the N-step Q-learning algorithm. And compare it to the Monte Carlo method. The aim was to see

Algorithm 6 Tabular n-step Q-learning and Monte Carlo RL

Input: Exploration parameter $\epsilon \in (0, 1]$, Learning rate $\alpha \in (0, 1]$, Discount parameter $\gamma \in [0, 1]$, Maximum episode length T
Initialization: A state-action value table $\hat{Q}(s, a) \leftarrow 0$ $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
while $budget$ **do**
  $states, actions, rewards \leftarrow 0$
  $states_0 \sim p_0(s)$; /* Initialize state */
  for $t = (0, .., T - 1$ **do**
    $budget \leftarrow budget + 1$
    $actions_t \sim \pi(actions|states_t)$
    $states_{t+1}, rewards_t \sim p(rewards_t, states_{t+1}|states_t, actions_t)$
  end for
  $update(states, actions, rewards)$; /* Update function */
end while
Return: $\hat{Q}(s, a)$

how the N-step Q-learning (that would start with only 1 as if it was the one implemented in Section 5 and end with a much larger depth) would perform against the Monte Carlo that tackles the problem by summing all the rewards in the episode.

6.3. Results

The results o the experiment in Section 6.2 are shown in Figure 7.

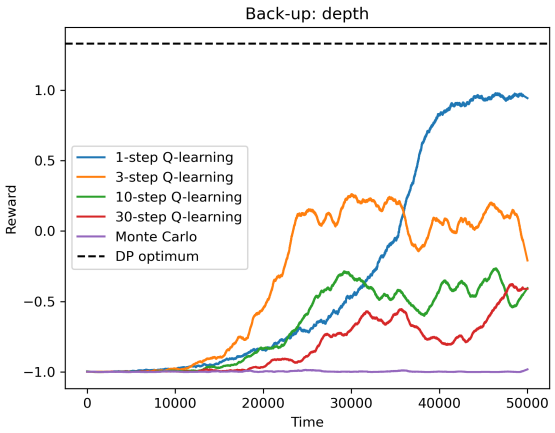

Figure 7. Q-learning and SARSA for different values of α

The first characteristic to notice in Figure 7 is that the Monte Carlo approach did not work at all. Concluding that, the sum of all the rewards of the episode is not a great way

to handle the Stochastic version of Wind Gridworld. It is possible to observe the same conclusion with the N-step Q-learning. Since augmenting the back-up depth (the higher the n the more close to the Monte Carlo approach) results in a worse performance overall. It learns slower and also has lower final reward. Similarly, when the n is smaller it has a better performance, since it behaves more like the results obtained in Section 5.3 with 1-step Q-learning.

7. Reflection

At the end of this assignment we can conclude several things that we have learnt through the process and the experiments performed. First is the big difference between Dynamic Programming and Reinforcement Learning, which is presented in every figure. DP knows everything about the model which allows it to arrive at the optimal policy much more faster and optimally, nevertheless, it has a big problem with scalability. If the model or problem to tackle is big enough, we might not be able to use DP due to the computational costs. On the other hand, RL is model-free (does not know it), meaning that it will take longer to achieve that optimal policy, but at the same time does not have the scalability problem as DP does.

For the exploration, both methods are useful and depends mostly on the problem to solve which is the best one.In our case the ϵ-greedy exploration performed a bit better due to the simplicity of the environment. but in another one where more exploration is needed it might be more interesting to use the softmax. Other and more complex methods can also be used, for example, methods based on swarms or evolutionary algorithms, However, ultimately, it depends on the problem.

The back-up information in Section 5 does have some benefits to it. To explain it clearly, in back-on policy the learning takes place by consistently backing up the value of the selected action back to the same behavior policy function that was used to select the action. Whilst, back-off policy refers to when learning takes place by backing up values of another action, not the one selected by the behavior policy.

As stated in the book (Sutton & Barto, 2018): This back-off policy methods makes sense when the behavior policy explores by selecting a non-optimal action, on-policy learning would then have to backup the value of the non-optimal exploration action. Off-policy learning, however, is free to backup the value of the best action instead, and not the inferior one selected by the exploration policy, not polluting the policy with a known inferior choice. Thus, in the case of exploration, off-policy learning can be more efficient, by not stubbornly backing up the value of the action selected by the behavior policy, but the value of an older, better, action instead.

Which all that in mind, we can not completely ensure that N-step Q-learnign is a fully off-policy method, since the first n reward are of course sampled from the current policy (and the target therefore mostly follows our behavioral policy).

Depth of target, additionally adds the N-step problem, which differs from the 1-step due to the ability to sum multiple rewards in a trace before we bootstrap. There are disadvantages to the Monte Carlo version. It generates a full random episode in each iteration, before it assesses the quality. (Value-based methods use a reward to select the next action in each time step of the episode.) Because of this, policy-based is low bias, since full random trajectories are generated. However, they are also high variance, since the full trajectory is generated randomly (whereas value-based uses the value for guidance at each selection step). The problems with that is:

- First, policy evaluation of full trajectories has low sample efficiency and high variance.Hence, policy improvement happens infrequently, leading to slow convergence compared to value-based methods.
- Second, this approach often finds a local optimum, since convergence to the global optimum takes too long.

Based on the experiments made in this assignment. we can assure the the Monte Carlo approach due to the previous stated problems, does not perform well. So the preferable method for this task should be N-step Q-leaning. Which propagates the information faster and is more likely to find the optima policy.

The problem with tabular RL algorithms is the curse of dimensioinality. Which is explained as the problem one would have when the environment is too large or has many possibilities. These method would need, in that case, a lot of computational power to achieve a result. For that, the use of Machine Learning methods like Deep Learning can help, when choosing the next action the policy should take. Or to simply reduce the amount of possibilities on the search space. If our RL algorithm is able to not only learn by error and trial but also with background of other similar experiments it would arrive much faster to an optimal solution.

References

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

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