Deep Reinforcement Learning
Assignment #1
Tabular Reinforcement Learning

Luca J. Barbera[1]

Abstract

This is a first introduction into and experimentation of the widely discussed field of Reinforcement Learning and its tabular learning basics. The optimal policy extracted from Dynamic Programming was compared to two action-selection policies that have to balance the exploration-exploitation trade-off of unsupervised learning. Furthermore, different ways of backing up the learning algorithms were be investigated: On- vs off-policy updating and the role of back-up depth. We gain a first outlook on why Neural Networks are able to improve Reinforcement Learning algorithms.

1. Introduction

Reinforcement Learning is at the frontier of artificial intelligence and has come to public popularity when outperforming humans at playing highly difficult games in unexpected ways. The first step to understanding and developing higher tier Reinforcement algorithms, however, is to look at a basic example and conduct experiments with it.

Here, we will look at an agent moving in a *windy gridworld* of 7×10 squares. It is a stochastic environment with discrete state space of 70 possible states and four possible actions (move up, right, down, left). Having full knowledge of the environment, we will implement Dynamic Programming, yielding the optimal policy that will later be used to measure the algorithm's learning performance. When the environment's every reaction is unknown, which we call *model-free reinforcement learning*, the agent will have to learn by trial and error. This consists of keeping a log-book type policy that tells the agent where best to go. Learning is then just the right combination of *exploiting* this knowledge and *exploring* to improve the knowledge. Here, we will look at two specific policies: Boltzmann and ϵ-greedy. We will investigate under which circumstances one outperforms the other, how much exploration is actually benificial and in what ways these algorithms will be useful going further. Next, we introduce SARSA and Q-learning, on- and off-policy methods of backing up information will be tested and their differences made clear. For more complex games and environments, a further look into the agent's future than just the immediate next step is crucial to estimate a certain state-action-combination's value. To this end, we will extend our Q-learning experiments to respect n more future step's values, up until the point where the whole trajectory (Monte Carlo) becomes relevant.

In the end the limit of tabular reinforcement learning becomes quite clear and an outlook into extension of our skill set using Deep Learning networks will be made.

2. Dynamic Programming

The goal in Dynamic Programming (DP) is not to learn by trial and error, but to calculate the best policy before even playing. The agent (or someone writing the policy-book for the agent) takes the full transition $p(s'|s, a)$ and rewards $r(s, a, s')$ tables (full information about the environment) to find the strategy that converges to the optimal policy. Since the agent only starts playing once all the *learning* (calculating, in fact) is done, it will follow the deterministic greedy policy: always perform the highest yielding action and do not explore.

We want to calculate the optimal $Q(s, a)$ table. The table shows us for each state-action pair (e.g. *from state 3, go left*) how much cumulative reward this pair will yield. This factors in not only the immediate reward but also future reward, that is, reward from the state that we will go to after the next state and so on.

2.1. Method

To this end, we go over each state-action pair and update the Q table according to the following update rule that uses the Bellman equation (1) for discrete state and action spaces

[1] S3705439, Leiden University. Correspondence to: Luca J. Barbera <l.j.barbera@umail.leidenuniv.nl>.

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

For a fixed state s and action a we sum over all other states s'. The Q-value will be the immediate reward we get from going to s' (i.e. $r(s,a,s')$) plus the best future Q-value we get in s', times some discount factor γ. This discount (when ≤ 1) will decreasingly respect the future Q-values the further we can go. Finally, we multiply their sum by $p(s'|s,a)$, the probability to get to state s' by acting a from state s. The probability will obviously be zero for most states s', as we can only move one cell from s doing action a, except for when the wind blows (see Fig. 1). We continue updating the Q-table as long as the values compared to the pre-update values change more than a certain threshold we set, as can be seen in Alg. 1. The algorithm does not cover the full code but is just an overview for the structure and the idea for the algorithm is oriented around the task given.

Algorithm 1 Q-table iteration in Dynamic Programming

```
Require: threshold $\eta \in \mathbb{R}_+$
  $Q(s,a) \leftarrow$ zeros
  while $\Delta > \eta$ do
    $\Delta \leftarrow 0$
    for all states do
      for all actions do
        $Q_{const.} \leftarrow Q(s,a) \in \mathbb{R}$
        $Q(s,a) \leftarrow \hat{Q}(s,a)$ {use update rule}
        $\Delta \leftarrow \max(|Q(s,a) - Q_{const}|, \Delta)$
      end for
    end for
  end while
  Return $\pi(s) = \arg\max_a Q(s,a)$ {extract optimal policy}
```

2.2. Discussion of Q-value iteration

After the first iteration rendered in Fig. 1 we see that only tiles close to the goal have a high value, as their probability to immediately reach the goal are high and thus get a large portion of the $+40$ reward. As we initialised the Q-values to 0, the future ($\max_{a'} Q(s',a')$) does not yet contribute and state-action pairs further away from the goal only get -1 rewards from their surrounding tiles. But even after only one iteration you see the arrows pointing to the top/right, which will be the direction in which the agent is preferably going to move.

Mid-way (after approx. 9 iterations) the right half is pretty much converged, while the left half still has to change (cf. Fig.2). High values move from right to left, from bottom

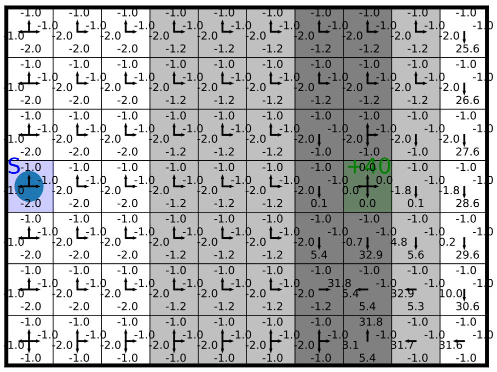
Figure 1. Q-table after one iteration. The wind blows with 80% probability on shaded tiles. The agent will be moved one tile upwards on light grey and two tiles upwards on dark grey.

to top (reverse final game direction) as far away states can only reach high values through future $Q(s,a)$ values that in turn depend on their future and so on.

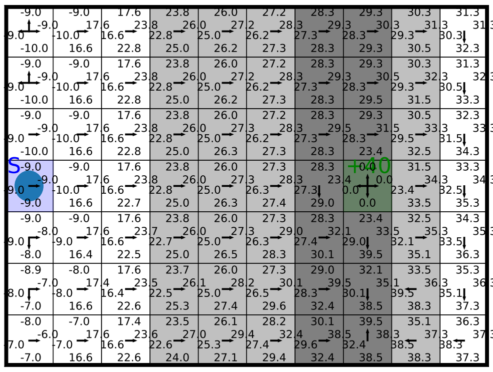
Figure 2. Q-table after 9 iterations (roughly mid-way).

Finally in Fig.3, all tiles have relatively high values. The right part has higher values and higher diversity inside one cell among different actions whereas the left part has only slight difference between actions per tile. But since the greedy action will pick the highest value, only a slight maximum has the same effect as a clear one.

The optimal value at the start is

$$V^*(s=3) = \max_{a'} Q(3,a') = 23.3, \tag{2}$$

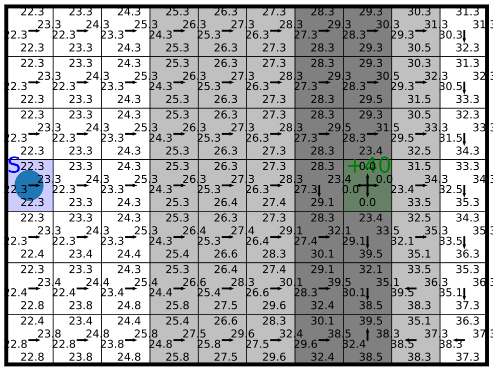

Figure 3. Q-table at convergence.

which means that the cumulative reward we can expect from starting in $s = 3$ is exactly that. This way we find the goal, get $+40$ while going $40 + -1 \cdot 23.3 + 1 = 17.7$ steps (with each a reward of -1) to get there. The last $+1$ is for the final step to the goal. As we can not go partial steps, the minimum amount of steps the agent has to take is rounded to 18. This leads us also to the average reward per time step which is the cumulative reward divided by the amount of steps needed: $23.3/18 = 1.29$.

Even though the goal state $s = 52$ is terminal, the Dynamic Programming algorithm converges, i.e. the maximum error of updating the Q-table undergoes a certain threshold. This is due to the fact that the four Q-values at $s = 52$ are constantly zero. The environment tells state $s = 52$ to stay in that state in 100% of the cases and receive a reward of 0 each time the agent *goes from 52 to 52*. The other term in Eq. (1) that could change the Q-value is the current Q-value's maximum of the next state, which, of course, was initialised to 0. The transition probability to all other states $s' \neq 52$ is 0%, which means those don't contribute to the update of Q either and $Q(52, a)$ remains zero. A simpler way to ensure convergence would be to just skip the update when $s = 52$ and then $Q(52, a)$ would be constant at 0 as well.

If we change the goal location to `[6,2]`, it takes more steps to converge and the agent moves down and right instead of up and right. Here it becomes clear again, that the Q-table converges first along the preferred trajectory, backwards from the goal. Tiles with fewer states in their future converge first and tiles further away from the goal that depend on the former converge last.

3. Model-free exploration

In this section we are going to take a different approach, that is, without knowing anything a priori about the environment. This time the agent will learn by actually moving around and updating the Q-table according to the state changes and rewards given by the environment. The game is going to work in episodes that terminate only when the goal is reached or the budget is exhausted. During this process, not all states of the space will be visited the same amount of times. But the agent will visit some state-action-pairs it knows yield a high value more often (exploitation of learned knowledge), while it still explores in other directions to learn something new (exploration of new possibilities). In the following part, the rate and type of exploration will be tested to get the best performance.

3.1. Method of exploration

We will implement two different stochastic policies that differ in the way the agent selects an action. We test the **ϵ-greedy policy**

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon \cdot \frac{|A|-1}{|A|}, & \text{if } a = \arg\max_{a'} Q(s, a') \\ \epsilon/|A|, & \text{otherwise} \end{cases} \tag{3}$$

versus the **Boltzmann policy** (also called softmax)

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in A} e^{Q(s,b)/\tau}}, \tag{4}$$

where $A = \{\uparrow, \rightarrow, \downarrow, \leftarrow\}$ is the set of actions and τ is the so-called temperature. A greater ϵ means more exploration and less exploitation of gathered knowledge, whereas the exploration in the Boltzmann policy is maximal when $\tau \to \infty$. In the latter case, the exponent goes to zero and the numerator is equal to 1, i.e. we divide 1 by $\sum_i^{|A|} 1 = 4$ which results in randomly picking one of the actions at all times. The softmax takes a vector of real numbers and returns a probability distribution over them, where large components of the vector get assigned to high probabilities.

3.2. Updating the Q-table

Having set up our action-selection policies, we now need our second major ingredient, updating the $Q(s, a)$ table. After performing action a from state s the agent lands in s' and receives reward r. The agent thinks back one state and attributes a value to it. In later sections this will be expanded to not only the previous state, but to a whole history of states (sequence of n-steps).

In a general case (1-step, n-step and ∞-step/Monte Carlo) we update the Q-table according to the **tabular learning update**

$$Q(s_t, a_t) \leftarrow \alpha \cdot G_t + (1-\alpha) \cdot Q(s_t, a_t), \qquad (5)$$

with the **back-up estimate** G_t

$$G_t = \sum_{i=0}^{n-1} \gamma^i \cdot r_{t+i} + \gamma^n \cdot \max_{a'} Q(s_{t+n}, a') \qquad (6)$$

The index t runs over the sequence steps the agent traverses. The simple fact that we include the learning rate α shows already that the agent is *learning* as opposed to all-knowingly calculating the best policy (see section 2).

It is important to note, that in the Monte Carlo (MC) case (see section 5) *bootstrapping* is dropped and $n \rightarrow \infty$. Adding bootstrapping means adding the (discounted) highest possible value at the final state of the sequence that was observed. In simple (1-step) Q-learning that is just the next state's maximum value. In n-step, first the next n (discounted) rewards will be added and then the sequence's last state's maximum value.

For a certain budget (amount of actions we afford to perform in the environment) we repeat along the following algorithm: Move to start location, select action according to one of the policies (constant over one experiment), retrieve reward and next state from environment and update the Q-table according to Eqs. (5)/(6). After the agent reaches the goal, everything besides the Q-table is reset. The core of this algorithm is listed in Alg. 2. As a measure of performance we look at the rewards obtained over time and multiple repetitions, as can be seen in Fig. 4.

Algorithm 2 Inner learning loop while budget is not yet exhausted

```
while budget do
    $a \sim \pi(a|s)$ {draw action from one of the policies}
    $r, s' \sim p(r, s'|s, a)$ {do action $a$}
    $Q(s, a) \leftarrow \tilde{Q}(s, a)$ {use $Q$-learning update rule}
    if $s$ is $terminal$ then
        reset environment
    else
        $s \leftarrow s'$
    end if
end while
Return $Q(s, a)$ {get converged $Q$-table}
```

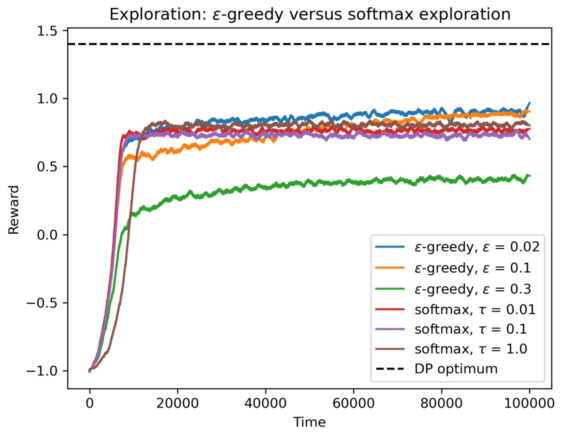

Figure 4. Exploration vs exploitation of two different action selection policies: ϵ-greedy (blue, orange, green) and Boltzmann (red, purple, brown). Curves are averaged over 50 repetitions and smoothed with a step window of 1000. The optimum of average reward per time step calculated after DP (section 2.1) is marked as a dashed line. The learning rate is $\alpha = 0.25$.

3.3. Discussion of exploration

The experiment shown in Fig. 4 includes several values for the exploration parameters ϵ and τ for both policies mentioned above. None of the policies actually converge towards the optimal policy that was predicted in DP (cf. section 2.) This is due to the lack of further tuning of hyperparameters such as ϵ, τ and most importantly the learning rate α. In the case of ϵ-greedy, a low percentage of exploration between 2-10% seems favorable as 30% exploration (green) takes a longer time to even get a positive average reward per time step. On the other hand will an exploitation of 98% (exploration of 2%) saturate rather quickly and seems to be surpassed by the 10% curve after 50 000 time steps. This makes sense as exploitation usually yields high rewards quickly after the goal was found, but has difficulty in exploring new, maybe superior strategies that a higher exploration rate will allow. In total, the optimal exploration rate lies around 10%, as it produces acceptable results quickly and has the tendency to increase performance in the long run. The softmax policy shows less diversity in final performance, be it high exploiting ($\tau = 0.01$) or high exploring ($\tau = 1$) and seems in general more reliable. Though, as said before, total exploration means $\tau \rightarrow \infty$, which is far from our upper bound of exploring $\tau = 1$. Due to the fact that softmax stagnates around 20 000 and ϵ-greedy still increases after 100 000 time steps, we deduct that $\tau = 1$ is exploring too little. In any case, this is only a first trial on hyperparameters ϵ and τ (let alone learning rate α) and further tuning will be required to start to approach the optimal policy.

4. Back up: On-policy vs off-policy target

The following part will compare different methods of backing up the Q-values: on-policy and off-policy. We have been using the off-policy back-up in the previous part due to the Q-learning update. We chose our actions according to ϵ-greedy or softmax, yet we bootstrapped the Q-table according to the **greedy** policy, that is, always going in direction of the highest Q value. The SARSA back-up does this differently: The agent actually goes two steps and bootstraps the current value according to the value the second step will yield. This way, not the highest yielding future value counts into the update, but the actual value that will be received by following our policy. Of course, off-policy agrees with on-policy for vanishing exploration, as every policy will become the greedy $\arg\max$-policy. One could condense the idea of both back-ups to:

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

where

$$a_{t+1} = \begin{cases} \text{chosen by current policy}, & \text{if SARSA} \\ \arg\max_{a'_{t+1}} Q(s_{t+1}, a'_{t+1}), & \text{if Q-learning.} \end{cases} \tag{8}$$

For SARSA that means that both terms of G_t are on-policy, whereas for Q-learning it is only the first (the reward comes from the step the agent took according to the policy). The chronology of the agent interacting with the environment follows a similar algorithm as the one in the previous section 3.1, Alg.2. The difference lies in the fact that not only the next state s_{t+1} is needed for the Q-table update, but also (in case of SARSA) the next action a_{t+1}. In practice this means to evolve the environment using the first action a_t and then to draw the second action before updating. Q-learning does not need the second action to update off-policy, but it will need to use it anyway to generate the next state after having performed the update.

4.1. Discussion of on-policy vs off-policy

When looking at Fig. 5, it becomes clear why the learning rate α is the most important hyperparameter. Even though we look at two different ways to back-up the Q-value, the difference due to changing α dominates. High learning rates yield quicker results but can be surpassed in the long run.

Q-learning learns from the values of a different policy than the one it uses to select an action. This makes the algorithm less stable and convergence takes longer. Q-learning learns immediately towards the highest yielding policy, whereas for SARSA Q-update and action-selection policy coincide and yield a more stable result.

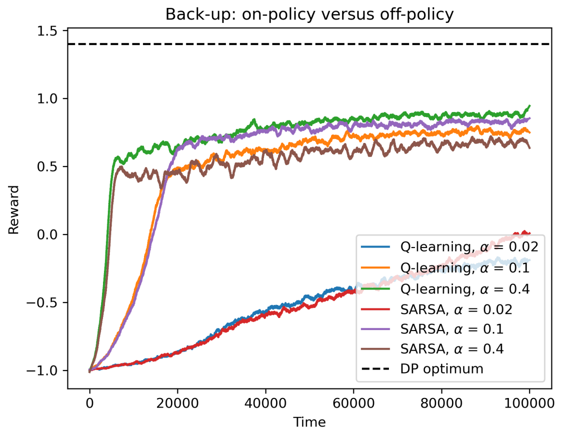

Figure 5. Reward vs time steps for comparison of on-policy SARSA with off-policy Q-learning table updates for different learning rates. Curves are averaged over 50 repetitions and smoothed with a step window of 1000. The optimum of average reward per time step calculated after DP (section 2.1) is marked as a dashed line.

5. Back-up depth: N-step Q-learning vs. Monte Carlo

As introduced in Eq. (6), we are now investigating the depth when backing up the Q-table update. The n-step algorithm is here the connecting method when going from simple Q-learning towards the Monte Carlo update algorithm. The Monte Carlo update is fully on-policy as its backup merely consists of rewards obtained by acting according to policy along the entire sequence. As seen in the previous section 4, Q-learning and with it also n-step Q-learning, is a method of bootstrapping the final sequence's value greedily, making this algorithm partially off-policy.

5.1. Method of finding the right depth

Dynamic Programming determined the optimal policy before even playing. Later, regular Q-learning and SARSA learned (updated Q-table) after each step. The n-step algorithm will play a sequence of n actions and afterwards learn from it, while the Monte Carlo method will play as long until it reaches the goal (or we interrupt it at a `max_episode_length`) and learns then.

Again, the algorithm will be working in range of a certain budget of how many steps the agent will interact with the environment. The agent plays a sequence of n or ∞ (MC) steps, all the while saving states, actions and rewards traversed (cf. Alg. 3). These arrays will then be used to update the Q-table according to Eqs. (5)/(6). Each step of each sequence reduces the budget by one.

Algorithm 3 Inner learning loop while budget is not yet exhausted

```
while budget do
    {reset environment}
    {collect sequence of $n/\infty$-steps}
    states, rewards, actions $\sim$ environment
    $G_t \leftarrow$ rewards, states, actions {bootstrap when $n$-step}
    $Q(s,a) \leftarrow \alpha \cdot G_t + (1-\alpha) \cdot Q(s,a)$
end while
Return $Q(s,a)$ {get converged $Q$-table}
```

5.2. Discussion of n-step vs Monte Carlo

As can be clearly seen in Fig. 6, 1-step Q-learning performs the best and performance decreases with length of the tried sequence before updating. Because of the way n-step and Monte Carlo learn, there only is a Q-table update every n or `max_episode_length`, as opposed to every step as in sections 3/4. Because the agent needs to take many steps before it can learn from these, convergence takes much longer. This relation is clearly not linear, as the 3-step algorithm started learning before the 1-step did, but is eventually surpassed by the latter. Longer sequence length also comes with more variance in the produced output — rewards depend on stochastic action choices and stochastic environment responses —, as can be seen by the oscillations in rewards per time step for larger n. On the other hand, higher variance also implies a decrease in bias from what has been learned so far.

The tendency of decreasing performance with increasing episode length culminates with the Monte Carlo method, where the played sequence is almost always `max_episode_length`=150 and with that much larger than the n-steps investigated. MC rarely escapes the -1 average that an interrupted sequence without the $+40$ reward yields. Even when it does, the agent does not seem to have learned a good policy and goes back to losing the game. Due to the long episode length, MC would need much longer to converge and the time scales should not be compared unconditionally. For this game, however, MC does not seem to be an appropriate choice and the right step length before each update should be in the low integers. Of course, none of the observed algorithms come close to the optimal reward per time step derived via Dynamic Programming in section 2, yet with further learning rate, exploration rate and other hyperparameter tuning the performance can certainly be improved in a similar time frame. So for further experiments, the 1-step Q-learning would be the way to go.

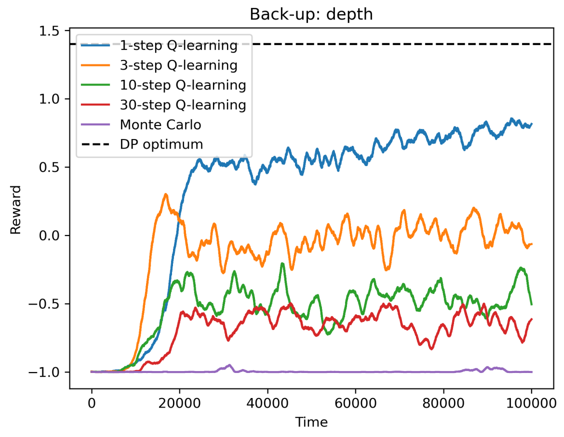

Figure 6. Reward vs time steps for n-step Q-learning and Monte Carlo update. Curves are averaged over 50 repetitions and smoothed with a step window of 1000. The optimum of average reward per time step calculated after DP (section 2.1) is marked as a dashed line. The learning rate is set to $\alpha = 0.25$.

6. Reflection

Dynamic Programming offers a simple introduction and understanding of the underlying dynamics of the environment. The algorithm to compute the optimal policy is straightforward and deterministic (even though the environment typically is not). Convergence is fast and the environment need not be prompted. But all this only works for a rather sub-complex state-action space. The curse of dimensionality quickly becomes an insurmountable challenge in DP. Going over every of the $256^{20\times20}$ state multiple times is impossible for a **single** RGB 20x20 image. In many problems where we want an agent to succeed, it is not omniscient and does not know all transition probabilities (or does not have enough memory to remember them).

That is why the agent needs to actually learn by doing and gather up a policy that delivers a satisfying strategy. We tested two exploratory policies: ϵ-greedy and Boltzmann. Tuning of the former is more accessible which makes experimentation easier, as exploration can be entered as a percentage ϵ rather than a temperature $0 \leq t \leq \infty$. Similar rewards per time step were achieved by both the policies, yet again the ϵ-greedy seems to be the preferred one, because performance with 10% exploration is still increasing after 100 000 time steps, whereas most Boltzmann temperatures lead to a stagnating performance before 20 000 steps. An alternative method to increase performance — other than exploration/exploitation tuning — could use Thompson sampling or add Dirichlet noise (1).

Later experiments tested off- versus on-policy, Q-learning versus SARSA algorithms. The way that SARSA uses one and the same policy for behavior (action selection) and learning (Q-table update) is straight forward and it converges timely and in a stable manner. Always updating along the behavioral policy can also soil it, though, for example when a bad (random) decision has been made by the policy. The alternative Q-learning approach immediately aims to learn and converge towards the optimal policy. This makes the way there less stable and thus increases the time until convergence, but generally delivers a reliable solution. Updating the behavioral policy using the greedy, well performing policy, though, can lead to a systematic overestimation of the Q-learning algorithm's performance. The continuation of the Q-learning idea to n-steps eventually leads to an algorithm that is mostly on-policy except on the very last term where bootstrapping happens off-policy.

Finally, the depth of the learning back-up was investigated. It became clear that for such problems where the number of actions needed to get to the goal is rather small, a large update depth is counterproductive. With the depth of each trajectory that is analysed comes the question of a bias-variance trade-off. Variance grew quickly with the length of sequences. An update after only a few steps (< 3), however, allows for faster information propagation and a much better performance that does not stagnate as quickly. Further parameter tuning will have to be done to the 1-step Q-learning until the optimal policy can be reached.

Most of the experiments could have used more time steps and further tuning. Running these algorithms for a small state space of 70 states, though, already required a high computational cost (especially n-step and MC). This shows that we are already running into the wall that is the curse of dimensionality. Of course, tabular reinforcement learning is straight forward and yields results without excessive tuning (only reinforcement learning and no extra deep network tuning). But soon memory will be full and we cannot afford to write a whole instruction manual what to for every single state.

We are going to need to group many states (and actions) together, extract overlapping features and approximate the best policy on that basis. The former two tasks are well performed by a Convolutional Neural Network (CNN), whereas the latter is done using a Deep Neural Network and all at an acceptable amount of memory. The agent needs to be able to learn a well performing policy without having visited all of the practically uncountable available states of larger state spaces.

References

[1] Aske Plaat. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022.