Deep Reinforcement Learning
– Assignment 1: Tabular Reinforcement Learning –

Lukas Welzel [1]

Abstract

In this essay I will use and compare various Tabular Reinforcement Learning methods. Every section is structured into ”Method”, where I explain the method and comment on expected behaviour, and ”Results & Discussion”, where I present and discuss the actual results from experiments. I include the relevant discussion and any extra work in the relevant sections themselves. I largely follow the algorithms as shown in the assignment and note when and how I deviate from them. I also largely use the notation of the assignment and Sutton & Barto (2018). The last section gives some general reflections that apply to all methods presented above. The appendix should not be relevant for grading. It explains an issue encountered with the base linear algebra libraries in my system.

1. Dynamic Programming

1.1. Method

I implement the Q-value iteration similar to Algorithm 1 in the assignment.[1] The `Q_value_iteration` function receives a `StochasticWindyGridworld` (`SWGW`) instance, a threshold η and the discount rate γ. It then crates a `QValueIterationAgent` instance and passes the discount rate and the shape of the `SWGW` to it. While the maximum error (or difference) Δ from iteration i to iteration $i+1$ is larger than the threshold η, the function iterates over every state $s \in \mathcal{S}^+$ and action $a \in \mathcal{A}$ in the value function $Q(s,a)$, initialized as the zero tensor with span $\mathcal{S}^+ \otimes \mathcal{A}$. Then, for each (s,a) in ${}_{\mathcal{S}}C_{\mathcal{A}}$, the new corresponding value $\hat{Q}(s,a)$, initialized as $Q(s,a)$, is updated by Equation 1, which uses a greedy local policy with a random tie-breaker.

[1]Faculty of Science, Leiden University, The Netherlands. Correspondence to: L. Welzel <`welzel@strw.leidenuniv.nl`>.

I think it is a crime to use the `subfigure` package in 2022. Not even the `subfig` package, but instead one which has been obsolete since 2002! The `subcaption` package exists for a reason. Luckily, the ICC in the Hague is pretty close...

[1]The code can be found in the repository available at `https://gitlab.com/lukas_welzel/deep-reinforcement-learning.git`.

$$\hat{Q}(s,a) \leftarrow \sum_{s'} p\left(s' \mid s,a\right) \cdot \left(r + \gamma \cdot \max_{a'} \hat{Q}\left(s', a'\right)\right), \tag{1}$$

where $p\left(s' \mid s,a\right)$ is the state-transition probability from state s to the next state s' when taking a, and r is the expected reward for that transition. We write $r\left(s,a,s'\right)$ as r since we have marginalized the reward probability distribution using Equation 2.

$$r\left(s,a,s'\right) \doteq \sum_{r \in \mathcal{R}} r \frac{p\left(s', r \mid s,a\right)}{p\left(s' \mid s,a\right)} \tag{2}$$

Each step the maximum error is updated using Equation 3.

$$\hat{\Delta} \leftarrow \max(\Delta, |Q(s,a) - \hat{Q}(s,a)|) \tag{3}$$

When the Q-value iteration has converged, `Q_value_iteration` yields the optimal value function $Q^\star(s,a)$ from which the converged optimal value $V^\star(s)$ and (greedy) policy $\pi^\star(s)$ can be recovered using Equation 4 and Equation 5 respectively. (Sutton & Barto, 2018; Moerland, 2022)

$$V^\star(s) = \max_a Q^\star(s,a) \tag{4}$$

$$\pi^\star(s) = \arg\max_a Q^\star(s,a) \tag{5}$$

1.2. Results & Discussion

Figure 1 shows the evolution of $Q(s,a)$, with the corresponding threshold and number of iterations. Initially the table is zero everywhere. As the number of iterations increases the algorithm essentially works backwards from the goal first increasing the values of cells from which the goal can be reached directly and then also increasing the values of cells from which other high value cells can be reached. The reverse is happening for cells far away from the goal which accumulate the negative step rewards from the surrounding cells. The goal have no value since there are no transitions out of the cell. The algorithm quickly finds the following (for the optimal policy after 2 iterations):

- The cells below the goal are have values since it can either walk into the goal or be pushed into it by the wind.

- The cell two steps below the goal has less worth than the surrounding cells since it the agent cannot reach the goal directly and cannot be pushed into it with the actions it has available.
- It is unlikely that the goal will be reached from the west due to having to step through wind for multiple steps.
- The cells to the east of the goal can be used to get to cells from which the agent can be pushed into the goal by the wind.

After two more iterations the algorithm finds that there is a small chance to get to the goal when coming from the south-west but it has also identified a way to get around the wind and to the goal from the south est after passing by the goal in the north. This gradient propagates towards to the start in the west during the next 8 iterations which causes an almost constant error. After 13 iterations there is a clear gradient from the start to the goal and the optimal policy and value function do not significantly change which causes the maximum error to drop until the algorithm terminates at $\Delta < \eta$. The converged optimal policy and associated values show that the algorithm tries to get to a position form which it can be pushed into the goal. It places very high values on cells that enable this. The agent can reach these cells either via the very east corridor without wind or via the southern rows. Since the stochastic wind is imprinted on the value function via $p\left(s' \mid s, a\right)$ the algorithm places higher value on the path that avoids relying on the wind not blowing. This path can be simply from the start by walking east which is indicated by the gradient in that direction. The converged optimal policy in the last panel of Figure 1 shows that the algorithm learns that the wind will either push it northwards towards the corridor or it can reach the goal directly. The goal cell itself always must have a reward of zero, and, in the implementation of `StochasticWindyGridworld`, only a transition into it (due to agent action or wind) returns the goal reward, which also terminates the run. Another solution to this, which would however change the dynamics of the system, is adding the action of standing still in one cell which terminates the episode when chosen on the goal cell.

$$\bar{N}_{steps} = r(s=G) - V^\star(s=S) + 1 \tag{6}$$

$$\bar{r} = \frac{r(s \neq G) \cdot (\bar{N}_{steps} - 1) + r(s=G)}{\bar{N}_{steps}} \tag{7}$$

$$\approx 1.034 \tag{8}$$

The converged optimal value at the start, which can be interpreted as the expected reward under the converged optimal policy when starting at S, is $V^\star(s = S = 3) = 18.3$. Since the reward when reaching G and for each step is known the expected mean reward per step can be computed using Equation 6.

2. Exploration

2.1. Method

I implement the Q-learning algorithm as similar to Algorithm 2 in the assignment. The algorithm proceeds initially similar to the one presented section 1, however with a `QLearningAgent` instead. For every time in a total budget the current state and the next action is selected, either using a ε-greedy (Equation 9) or Boltzmann policy (Equation 10).

$$\pi(a \mid s) = \begin{cases} 1.0 - \frac{|\mathcal{A}|-1}{|\mathcal{A}|}\epsilon, & a = \arg\max_{b\in\mathcal{A}} \hat{Q}(s,b) \\ \epsilon/(|\mathcal{A}|), & \text{random } a \end{cases} \tag{9}$$

$$\pi(a \mid s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{b\in\mathcal{A}} e^{\hat{Q}(s,b)/\tau}}, \tag{10}$$

where ϵ and τ are the relaxation parameters of the policies which lead to (greedy) exploitation when small ($\simeq 0$) or exploration when large ($\epsilon \simeq 1$ or $\tau \gg 1$.) and might change when Q-learning (annealing). The environment is then propagated by one step which yields a reward and the next state s'. The value function is then updated by Equation 11.

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

where a subscript $_t$ indicates the current time-step in the episode, α is the learning rate and the back-up estimate G_t is given by Equation 12.

$$G_t = r_t + \gamma \cdot \max_{a'} \hat{Q}\left(s_{t+1}, a'\right) \tag{12}$$

Finally, a check is performed if the terminal state has been reached. If it has been reached, the environment is reset and a new episode is started, otherwise the next (time-) step is taken. Once the budget is exhausted Q-learning terminates immediately without finishing the episode. A maximum episode length is implemented for this and the following methods to avoid infinite loops and episodes that do not end. Since the environment is a gridworld this technically violates the Markov property of the decision process, and to be rigorous the time step needs to be included in the state. Since I do *not* want to be rigorous I simply set the maximum episode length to be large compared to the expected episode length, which is an accepted solution to this problem.

I implement several annealing methods for both ϵ and τ via a `AnnealScheduler` class:

- `linear_anneal`: A version of the provided function that can be vectorized.

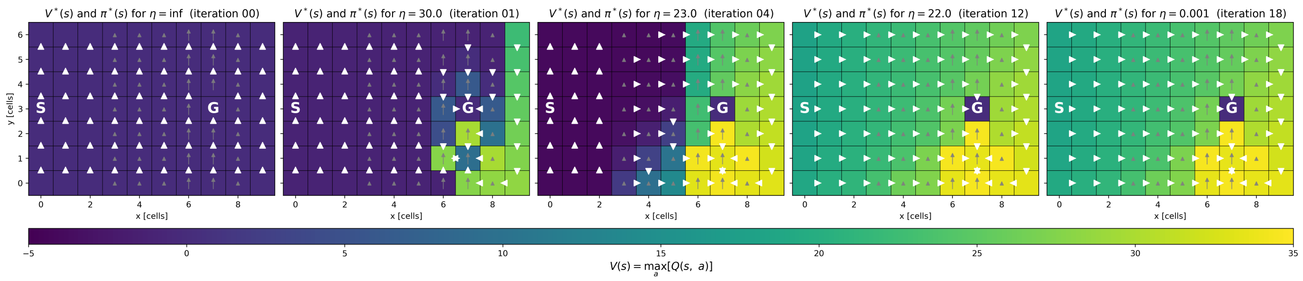

Figure 1. Evolution of the converged optimal value $V^\star(s)$ and optimal (greedy) policy $\pi^\star(s)$ during (representative) value iterations. S shows the starting position, G is the goal position, the white arrows (triangles) are the best action at state s with value $V^\star(s)$ which is shown by the fields' color. and the gray arrows the wind strength. Q-value iteration converges after 19 iterations to the optimal policy, which is indicated by the white arrows, with a threshold of $\Delta = 0.001$ and discount rate $\gamma = 1$ (no discount). As the algorithm iterates it learns "backwards" from G which fields have high values (from where it can reach G) until it finds a gradient from S to G and then converges quickly to the optimal value function $Q^\star(s, a)$ and $\pi^\star(s)$. In this stochastic, windy Grid World it learns that it is very unlikely to reach G directly and finds that it is best to reach G are by being pushed into it by the wind. This is because it does not discount future rewards $\gamma = 1$.

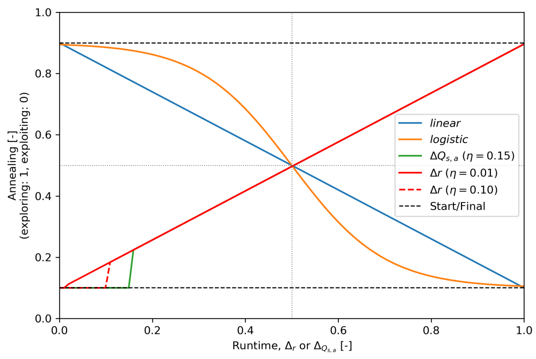

Figure 2. Annealing schedules. The x-axis shows (fractional) runtime for both the linear and logistic schedule whereas it shows the difference between this and the previous step for both the value function and reward difference scheduler. The y-axis shows the annealing status where 1 is random exploration and 0 is full exploitation (greedy). The initial and final values for the annealing are shown as dashed black lines. Whereas the linear and logistic anneal depend only on the progression through the timesteps the other anneal schedules are based on the gradient of the learning and, hence might change non-linearly. Especially the reward difference anneal is susceptible to feedback.

- `logistic_anneal`: Annealing based on the logistic function $\mathcal{L}(x) = \left[1 + e^{-k(t-t_0)}\right]^{-1}$, where k is a scaling parameter. The annealing essentially asks the question if the algorithm should explore or exploit. This is a binary choice and hence we would expect the outcome to the the form of a logistic function.

- Linear `q_error_anneal`: Scheduling the anneal based on the difference (error) between $\hat{Q}(s_{t-1}, a_{t-1})$ and $\hat{Q}(s_t, a_t)$. Initially the error will be large which is an indicator that the environment is not well explored, when the error becomes small the environment is well known and the algorithm should start exploiting to maximize rewards. To smooth the anneal this is tracked in a buffer over some N steps.

- Linear `r_diff_anneal`: An anneal schedule similar to `q_error_anneal` which tracks the differences of the rewards instead. Similar to before a small change in mean rewards might indicate that the environment is well known. However, since the mean reward is also a function of ϵ or τ this is risky; if the exploration is completely random the rewards will have the same variance for any $\hat{Q}$. Hence, $\epsilon > 0$ or $\tau > 0$ so that this issue can be avoided. One advantage of this over above method is that it only needs to keep track of very few parameters (mean reward) instead of past $\hat{Q}$.

For all schedules annealing is performed after each episode and maximum/minimum anneal values can be specified. Figure 2 shows examples of the annealing schedules. Both the value function and reward difference anneal can be supplied with a threshold η. When the difference becomes smaller than this threshold the algorithm will start exploiting as much as possible. However, since these algorthems continue to monitor the performance they can start exploring again after they have reached "full" anneal. If this anneal was due to a non-optimal policy the stochastic environment can lead to a new phase of (small) exploration if it changes the value function of mean reward more than allowed by the threshold. This also gives the algorithms the ability to respond to a change in the environment dynamics after annealing is nominally completed. Naturally, the annealing methods can be stacked or otherwise combined to enable more complex annealing.

2.2. Results & Discussion

Naturally, the Q-learning perform worse than the Q-value iteration algorithms, both in absolute performance and performance per CPU-time. This is because the agent does not have a complete model of the environment i.e. needs to explore the environment manually. The performance of the different algorithms with varying hyper-parameters is shown in Figure 3. Greedy algorithms perform better in this experiment. It is nevertheless noteworthy that the environment which with the agents interact is relatively simple and small. For more complex environments the greedy policies might converge to local minima that less greedy policies would not as easily get stuck in. Naturally, given enough time, the algorithms should still converge to the optimal policy. Which method is generally preferable depends on all, the use-case, environment and resources.

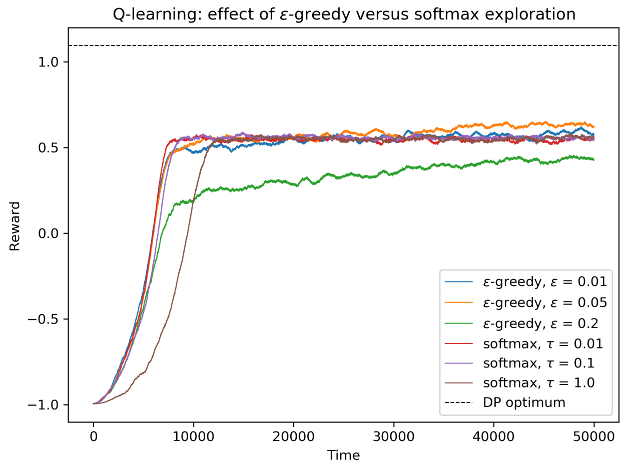

Figure 3. Differences between the policy for Q-learning. Generally, greedy policies perform well, both learning quickly and reaching a relatively high mean reward. Less greedy policies learn both slower, but reach a similar mean reward under their converged policy. The time this takes for a Boltzmann policy with $\tau = 1$ is larger than the maximum budget allotted. While the mean reward becomes quickly positive (after max. 10^4 steps) for all Q-learning algorithms, it is at best $\approx 60\%$ of the mean reward from value iteration. This is because the environment is unknown and needs to be explored step-by-step.

While in above experiments both ϵ-greedy and Boltzmann (softmax) policies perform similarly well, especially for greedy policies, the softmax policies are sometimes preferable in practice if performance is the priority. This is because both policies sometimes sample the non-optimal action under the current policy and a ϵ-greedy policy then chooses from the other options with equal probability, no matter if there is a difference in their expected return. This is a problem that softmax avoids by scaling the probabilities proportionally to their expected rewards. Nevertheless, both policies rely on only one parameter (neglecting annealing) and confidently estimating a good ϵ is in practice easier than the temperature for softmax[2] according to (Sutton & Barto, 2018). Additionally, the entropy of a softmax policy depends on the magnitude of the rewards and hence knowledge of the environment is required to confidently estimate τ, whereas the entropy of an ϵ-greedy policy is is independent on the return magnitude. There are several approaches which soften the impact of or exploit this property of the policies such as Entropy-Augmented, and/or Entropy-Regularized policies (Ahmed et al., 2019; Lee, 2020) as well as Maximum Entropy (MaxEnt RL) such as Mellowmax policies (Asadi & Littman, 2017). When using SARSA with a softmax policy Asadi & Littman (2017) show that convergence (using tabular updates) is not guaranteed in all circumstances. They propose a differentiable softmax policy with a state dependent temperature that conserves entropy and, and like the other examples above, makes specifically exploration more robust (Eysenbach & Levine, 2021). However, this also demonstrates an important advantage of the regular softmax and especially ϵ-greedy policies; their simplicity and generality.

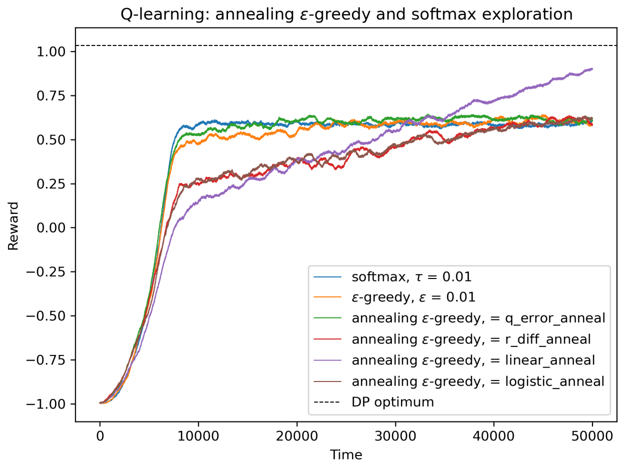

Figure 4. Q-learning with varying anneal schedulers. All runs with annealing used an ϵ-greedy policy, a buffer with a depth of 10, an initial value of $\epsilon = 0.3$, a final value of $\epsilon = 0$, a threshold of $\eta = 0.02$, and a percentage of 0.5. The `q_error_anneal` performs similar to the best runs that do not use annealing. The linear anneal outperforms all other methods w.r.t. the mean reward at the end of each repetition. The initial learning rate of the linear, logistic and reward difference anneals is low and similar to an ϵ-greedy policy with $\epsilon = 0.2$.

Figure 4 shows the performance of Q-learning with the annealing methods. In general the performance is not much better or worse than the methods without anneal, expect for the linear anneal which, while initially returning low mean rewards, can use the additional time spend exploring to reach an exceptionally high mean reward. Since the

[2]Though there are no conclusive comparative studies.

annealing is set to zero at the end of the budget the final performance might even be better than shown in the graph because of the smoothing. Increasing the budget might lead to a converged policy very close to the true optimum from Q-value iteration. This might be true for the reward difference and logistic anneal as well, however it seems likely that they behave somewhat asymptotically to the other methods near the very end of the budget. The small difference between the methods is probably due to the way in which they are coupled to the performance of the Q-learning itself. From Figure 3 one can see where the Q-learning stops improving. This is reflected in the annealed Q-learning as well since the Q-value and reward difference algorithm switch to a greedy policy and also stop improving. I am unsure why the logistic anneal performs worse than the linear anneal. I suspected it would both explore and exploit better, however it potentially misses out on properly utilizing gradients during learning (around 20 000 steps), when when ϵ starts changing rapidly. That this is also the point where the linear anneal starts outperforming the logistic anneal supports this interpretation.

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

3.1. Method

I implement SARSA similar to Algorithm 3 in the assignment. The differences from the implementation of Q-learning is the SARSA back up estimate, shown in Equation 13. Instead of making the best possible action (under a greedy policy) the target we use the actual next action we take to inform the value function.

$$G_t = \begin{cases} r_t + \gamma \cdot \hat{Q}\left(s_{t+1}, a_{t+1}\right) & \text{if } s_{t+1} \text{ is not terminal} \\ r_t & \text{otherwise} \end{cases} \tag{13}$$

This also requires taking the next action before the value function can be updated so that the order of operations slightly changes. The back-up estimate returns zero when the terminal state is reached. Instead of implementing this with an additional comparison I exploit boolean operations in `python` and multiply the back-up estimate with (**`not`** `done`), where `done` is true if the agent has reached the terminal state and false otherwise.

Table 1. Main difference between Q-learning and SARSA.

	Q-learning	SARSA
a'	π	π
a' for updating $\hat{Q}_{s,a}$	$\arg\max_{a'} Q_{s,a}$	π

The primary difference between SARSA and Q-learning, shown in Table 1, is how they chose to account for reward from future actions, SARSA return rewards from on-policy actions i.e. the expected reward when continuing to follow the current policy. Q-learning instead returns rewards from off-policy (here a greedy policy with $\epsilon = 0$). Naturally, the policies are the same when $\epsilon = \tau = 0$ i.e. greedy.

3.2. Results & Discussion

As shown in Figure 5 a smaller learning rate leads to slower learning, but also more stable learning and exploitation. Q-learning initially learns slightly faster than SARSA, however it is eventually outperformed in terms of mean reward until both methods have sufficiently relaxed. The algorithms with a large learning rate become somewhat unstable when they have converged. This is especially true for SARSA because it only converge to an optimal solution if the policy that generated the optimal policy is continually followed. Since there is still some randomness in the policy ($\epsilon, T \neq 0$) this is not guaranteed. This is reinforced by the short buffer which just includes one past value. Annealing the learning rate during the exploration phase (with a completed anneal of $\alpha_t(t > t_{explore}) \leftarrow \alpha_{min}$, where $\alpha_{min} \ll 1$ when exploitation should begin), as well as the ϵ or τ can alleviate this issue and assures more stable exploitation.

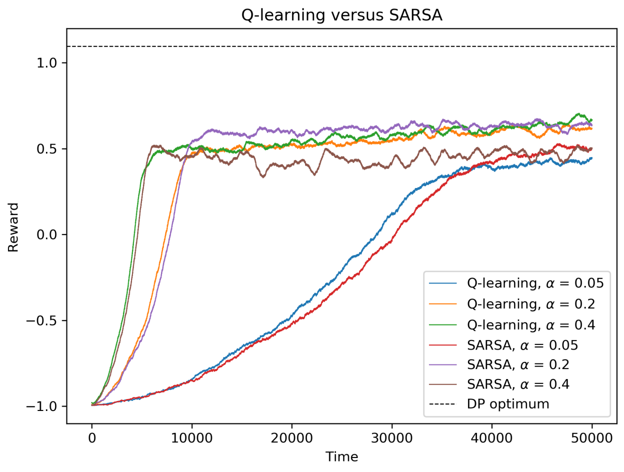

Figure 5. Q-learning compared to SARSA with varying learning rates. Both use a ϵ-greedy policy. Large learning quickly converge to a policy with relatively high reward but are unstable when exploiting the found strategy. For algorithms with a low learning rate it is the reverse.

What method is preferable depends on what the requirements on the algorithm are: In case fast learning is important (e.g. to quickly learn how to avoid expensive errors) then a large learning rate is important. On the other hand, if a stable exploitation is more desirable then the opposite is true. In a trade-off between the two this experiment shows that SARSA with $\alpha = 0.2$ converges reasonably fast, and somewhat outperforms the other algorithms for a while due to it being on-policy and the stochastic wind. SARSA is also preferable if there are severe consequences to failing when learning i.e. a wrong actions at the wrong step breaks the agent in the real world environment. While SARSA still includes some randomness due to the policy we chose we avoid making the same (potentially) fatal error many times, even if it means that our converged policy might not be completely optimal. Q-learning would be preferable if the performance of the agent during learning is not important and a switch to an optimal greedy policy when beginning exploitation.

4. Back-up: Depth of target

4.1. Method

I implement the n-step methods similar to algorithms 4 and 5 in the assignment. Again, the tabular update of the value function, given in Equation 11, is used for both the n-step Q-learning and Monte Carlo (MC) methods. For these methods the depth of the back-up estimate is increased to more than one, similarly to what has been described in section 2 to smooth the annealing process. This requires that the tabular update is applied after the episode has concluded, since the trace that has been taken is used in the back-up estimate. The back-up estimate for n-step Q-learning for timestep t is given in Equation 14.

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \max_a Q\left(s_{t+n}, a\right), \qquad (14)$$

where n is the depth, for which $0 \leq n < T_{episode} - t$ with $T_{episode}$ the total episode length. This method is practically off-policy, with the exception of some relaxation time after the value function was initialized where the current (initial) policy is sampled. If all rewards from the trace are sampled for the back-up estimate the update is a MC update, where G_t is given by Equation 15.

$$G_t = \sum_{i=0}^{T_{episode}-1} (\gamma)^i \cdot r_{t+i} \qquad (15)$$

My method differs from the algorithms shown in that they accumulate states, actions and rewards into pre-initialized arrays, parts of which are then passed to the tabular update function depending on the length of the episode. The rewards are accumulated using the already existing array for all rewards of a run. The back-up estimate for the MC method is directly calculated, as shown in Equation 15, instead of recursively as shown in the assignment algorithm box.

These two methods highlight a common issue for predictive algorithms; the trade of variance and bias in the estimate/sample. It is desirable that both properties of the predictor are low, however in almost all cases decreasing one results in decreasing the other. This is because a low bias is necessary to find regularities in the input (avoid under-fitting) but low variance is required to be able to generalize the algorithm beyond the training set (avoid over-fitting to noise). Hence, the improving the performance is typically impossible beyond some point of generalized error including noise. The reverse is not the case, i.e. a algorithm can have both high variance and bias.[3] In RL these errors step from the bias the learning algorithm introduces to the estimator and the limited data that can be trained on which relates to variance. In the case of n-step methods this trade is performed by increasing the depth of the algorithm and hence increasing the variance.

I have not tested annealing of the depth of target during the runtime but it would be an interesting experiment to optimize the depth during a run. This sounds a little too much like meta-learning for the first assignment though. Since it seems like there is an optimum depth at different times the trade towards the end would be of long-term stability against adaptivity.

4.2. Results & Discussion

In this experiment all (true) multi-step methods perform worse than the previous algorithms, both in stability and mean reward, as well as time-steps until initial learning was complete. This is mainly because they are unsuited for the environment which only gives positive rewards in a very specific cell with negative rewards otherwise. This means that, seeing the gridworld as a field with the rewards in the cells, the field is entirely flat except for the goal state, hence there are no gradients (in the environment reward) for these functions to detect. This further increases the high variance inherent to methods with a large depth, and in the extreme case of MC updates cripples the method beyond any benefit that can be compensated by its low bias. The n-step Q-learning methods learn at different rates with the 3-step method outperforming others in this experiment. This is because they propagate information at different speeds, decreasing with increasing depth. This implies that there is an optimal depth for the largest initial Q-learning rate, in this case between one and five. The single step method is however best at exploiting a policy once it is found and returns the highest mean reward when learning is completed hence, for this problem it seems to be the most suitable. Nevertheless, given an infinite amount of time (and memory) MC updates can converge to the optimum given that the learning rate and policy (and discount rate to increase the rate of convergence) is annealed during the run. As is shown in this experiment and the further experiments below, this is only possible theoretically and it seems that n-step Q-learning, while not mathematically rigorous, will converge

[3] This is what one could often call a bad estimator.

to a policy close enough to optimal much faster.[4]

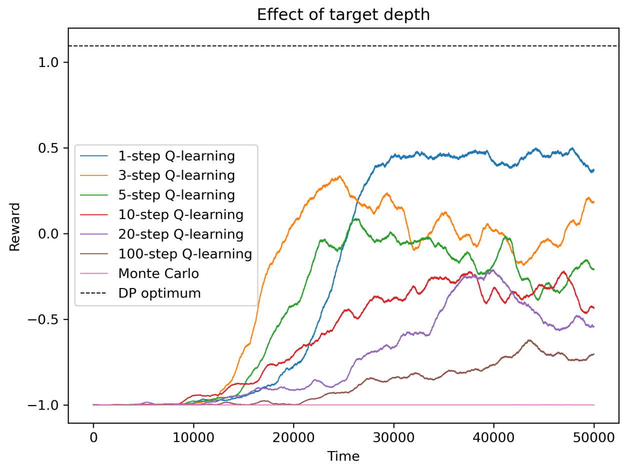

Figure 6. N-step Q-learning compared to MC updates with varying depth of target. Both use a ϵ-greedy policy. N-step Q-learning performs worse the larger the depth of target becomes. For depth of targets larger than 5-10 steps the algorithm does not learn a policy with a positive reward. The MC method is reliably unable to learn a policy that is better than random choices with a total budget of 5×10^4 steps since it is not able to construct a sufficient gradient when exploring. Anecdotally, it is also unable to find a policy with a mean return of larger than zero with a total budget of 10^6 steps.

I was particularly surprised by how unsuited the MC update was. However, this is reasonable for this environment since the updates are episodic, i.e. the algorithm does not learn during an episode and taking similar paths with just slightly different end states (i.e. one is terminal, the other not) results in very different values being assigned to the cells.

Hence, I changed some aspects of the MC update; 1) initializing the value function to -5 instead of zero to give a larger gradient around the goal, 2) reducing the discount rate to 0.6 to limit the effective depth and most importantly 3) initializing the agent at a random starting position. This improved the performance somewhat and, again anecdotally, the algorithm was able to archive a mean reward between 0.5 and 0.6 after 5×10^5 steps in three independent runs. Nevertheless, the policy was still unstable up until the budget ran out after 10^6 steps for each of these trials. Furthermore, I was curious as to why there is just minimal mean rewards, even at the end of the episode (the mean reward in the last $10\,000$ steps was ≈ -0.98, whereas the mean reward for the last 10^5 steps was ≈ -0.8). It seems like the method finds an optimal policy but then loses it again after some $100 - 1000$ episodes. This might be again due to the inherent instability of these methods on a very flat reward field.

[4]Up to some precision (which might be low as can be seen from this experiment) and under the same assumptions as for the MC method.

5. Reflection

5.1. DP versus RL

As is shown above, dynamic programming has significant advantages over RL, however also requires perfect knowledge about/a model of the environment, i.e. the transition probabilities and expected rewards, and can be more computationally expensive (also see subsection 5.2). In contrast RL methods only need to be able to interact with (sample) the environment through an agent. Hence, DP methods need access to $p(s', r \mid s, a)$ and can be most easily used if the problem/state space fits into memory and it can be iterated over by the actual machine. If the state space requires more memory than available it can sometimes be reduced by modifying it at the danger of learning non-optimal policies.

I would have liked to experiment more with the annealing schedulers, especially with optimizing them. Above I have only annealed ϵ, however varying the learning rate and discount rate during a run might also improve the performance. In this case however, that would already be a large parameter space consisting of 10^{10} combinations.

$$\mathbb{N} = (N_{schedulers})^{N_{hyper-parameters} \cdot N_{policies} \cdot N_{methods}} \tag{16}$$

$$\approx 10^{10} \tag{17}$$

Hence, it is completely impossible to test the effects of each of these combinations and hopefully we will learn a better way during the meta-learning lectures.

5.2. Curse of Dimensionality

In this work I have used tabular RL to estimate optimal policies. The state-action space of the environment is very small and hence it can be stored in working memory. This means that we are able to randomly access any value we require for the method at any time. This means that the algorithms can quickly converge and for DP even iterate over the whole state-action space. However, as soon as the state(-action) space becomes large we will no longer be able to quickly access data points. This becomes especially true if the dimensionality of the problem increases as modern memory relies heavily on data sorting and organization which becomes increasingly impossible even if the data set fits into memory. Hence, TRL is unsuitable for large datasets in general and high dimensional ones in specific. Above argument also applies to other RL methods to some extend. These methods however, often do nor rely on sampling and can therefor deal with much larger data sets. Nevertheless, even they will encounter some limit, and hence suffer from the ”curse of dimensionality”, shown in Equation 20.

Storing non-sparse tabular data requires some sort of storage volume, e.g. some space in (V)RAM. Generally, the required storage volume increases exponentially with the dimension of the data set. This means that data sets with a high number of irreducible dimensions must be sparse in practice. This can be viewed from different perspectives:

- In terms of time or equivalently work, the curse means that the time to search and store data increases exponentially (Donoho et al., 2000).
- In terms of density it means that all other things being equal, a data point in N dimensions has more neighbours within a unit n-ball than a point in $N + 1$ dimensions.
- In terms of distance, it means that the higher the dimensionality of the data set the more similar the distances between any two data points tend to become. See Equation 18 and Equation 19 which shows that the volume of N-dimensional spheres tends to be "concentrated" near its outer surface, e.g. compare a circle and sphere.

$$V_{\mathrm{d}}(R) \propto R^{\mathrm{d}} \tag{18}$$

$$V_{\mathrm{d}}(1) - V_{\mathrm{d}}(1-r) \propto 1 - (1-r)^{\mathrm{d}} \tag{19}$$

$$\therefore |\mathbf{X}| \sim \exp(\mathrm{d}(\mathbf{X})), \tag{20}$$

where V_{d} is the volume of an n-ball with dimension d, R, r are the outer radius of the n-ball and some radius smaller than unity respectively and $\mathbf{X}$ is some data set with size $|\mathbf{X}|$ (Bishop & Nasrabadi, 2006). This problem can approached from different ways; the most obvious one is to reduce the dimensionality of the data and apply a K-nearest neighbors algorithm to prepare the data set for efficient search. Dimensionality reduction encompasses a number of methods that select and extract features/correlations from high dimensional data and preserve them as much as desired in lower dimensional data. This is a trade-off between dimensionality and error or feature loss. Naturally, deep learning methods have an advantage in this regard since they can sample data points individually and do not require the whole data set at all times like TRL methods. (Bishop & Nasrabadi, 2006) There is also the other side of the curse, the "blessing of dimensionality". This refers, among other interpretations to the distances between two points becoming increasingly similar (i.e. approach the average distance of point pairs in the data set) which can be exploited (Donoho et al., 2000).

References

Ahmed, Z., Le Roux, N., Norouzi, M., and Schuurmans, D. Understanding the impact of entropy on policy optimization. In *International conference on machine learning*, pp. 151–160. PMLR, 2019.

Asadi, K. and Littman, M. L. An alternative softmax operator for reinforcement learning. In *International Conference on Machine Learning*, pp. 243–252. PMLR, 2017.

Bishop, C. M. and Nasrabadi, N. M. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Donoho, D. L. et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.

Eysenbach, B. and Levine, S. Maximum entropy rl (provably) solves some robust rl problems. *arXiv preprint arXiv:2103.06257*, 2021.

Lee, D. Entropy-augmented entropy-regularized reinforcement learning and a continuous path from policy gradient to q-learning. *arXiv preprint arXiv:2005.08844*, 2020.

Moerland, T. Assignment: Tabular reinforcement learning. Course: Reinforcement Learning, Master CS (Leiden University), Feb. 2022.

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

6. Appendix

6.1. Linear Algebra Library Issue

My specific python setup and machine have an issue with the Math Kernel Library (MKL) that is used by e.g. SciPy. This is sadly not (easily) fixable through python (virtual) environments or anything else since it is an issue of the base linear algebra libraries from Intel (such as BLAS and LAPACK) and my specific setup. For this assignment this means that I cannot use the `mode=interpolate` option of the `scipy.signal.savgol_filter` function. Instead I resort to the `mode=mirror` option which does not have this problem. Since both the window for the filter and the total number of timesteps is large this should not result in an significant difference.

6.2. .h5 File Funkiness

Depending on the permissions of the console you use it might be that a `.h5` file cannot be automatically created by `h5py`. In that case it should be sufficient to pop a text-file into the working directory and rename it to `runs_normal.h5`. I am trying to catch all the exceptions this might throw, but in the end one might still slip though on your OS.