UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ROBOTICS
AND COGNITIVE SYSTEMS

Course: Reinforcement Learning (SS2020)
Prof. Dr. Elmar Rueckert & Honghu Xue, M.Sc.
Assignment Report Nr. I
Date: 10. Juni 2021

Team Nr. 7
718651 RICARDO SARAU
737548 FAZLI FARUK OKUMUS
737551 MEVLUDE TIGRE
Contact: ricardo.sarau@student.uni-luebeck.de
fazli.okumus@student.uni-luebeck.de
mevluede.tigre@student.uni-luebeck.de

Task I: Theoretical understanding on multi-step TD learning and Eligibility Trace

1. **1)** The differences between one-step and multi-step TD algorithms regarding the q-update rule are the following: In one-step TD algorithms the update of a q-value for a state-action pair happens after the q-value of a state-action pair at the next time-step is computed. This is then done for each step in an episode with the following formula:

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \tag{1}$$

For multi-step algorithms on the other hand the q-value only gets updated for specific time-steps, which depends on the number of steps n of the algorithm. The agent takes actions to transition between states each time-step like in one-step TD algorithms and the rewards of each of these time-steps are stored. If the agent reaches a certain time-step t, then the q-value at the time-step τ gets updated following the rule $\tau \leftarrow t - n + 1$ until $tau = T - 1$ with T being the final time-step of an episode. To compute the q-value of all the time-steps in between the current time-step t of the agent and the time-step τ, where the estimate happens, the following formula is used:

$$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + alpha[G - Q(S_\tau, A_\tau)] \tag{2}$$

2) The n-step return G adds up the rewards over the defined time frame n and multiplies them with the decaying factor γ like shown in the following formula:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{3}$$

The λ-return is the return in the TD(λ) algorithm, which is used to average the updates over n-steps. This averaged return G_t^λ is computed in a way that all weights λ are summed up to 1. Formula for the λ-return:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\text{inf}} \lambda^{n-1} G_{t:t+n} \tag{4}$$

2. **1)** Importance sampling ratio is the relative probability of trajectories occurring under policies of the target and behavior polices, which are used to weight returns. This is necessary for off-policy learning, because the estimated returns are calculated under the behavior policy, but to obtain the values $v_\pi(s)$, we would need the returns under the target policy. Importance sampling can fix this issue, since the ratio transforms the returns to have the right expected value.
2) To get the right estimated return over n-steps in off-policy learning, the importance sampling ratio has to be factored in. This is obviously effecting the Q-Value update, since the estimated return is necessary to calculate the update as can be seen in the following formula:

$$G_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t + \alpha p_{t+1:t+n}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t] \tag{5}$$

Nearly the same applies to the Sarsa version, but instead of using $p_{t+1:t+n}$ as the importance sampling ratio $p_{t+1:t+n-1}$ is used. The reason for this is that although every possible action is taken into account in the last state, the chosen action doesn't need to be corrected.
3) Possible problems of importance sampling are that the updates to the Q-Values can be of high variance. One idea to counter this is to lower the step-size parameter, but that would inevitably result in a slower learning process.

3. Bias in the instance of reinforcement learning refers to the necessary assumptions to estimate the return G and how that effects the following estimates on the trajectory of an agent. Variance on the other hand refers to the amount of noise in form of all the possible states and actions in the return estimates. This is a trade-off mostly when it comes down to the question if Monte-Carlo or TD learning should be used. Since Monte-Carlo learning computes the return over all states and actions to get the value function, there aren't much assumptions made if any and the bias is low. This on the other hand leads to a higher variance, because even returns from actions are taken into account that differ highly from the actions in the optimal value function. In case of one-step TD learning variance is low, because the return G_t only depends the next reward R_{t+1}, which leads to less noise. The trade-off in order to calculate the estimated return, only the estimated value of the next state $V(S_{t+1})$ is looked at, which is an assumption on the optimal value function that highly effects the return so the bias is higher. This bias is decreased for 10-step TD learning, since the reliance on the value estimate of the next step is lowered with the number of steps we look into the future, though that increases the variance as we introduce more noise with the new states and actions we have to account for.

4. Depending one what you choose as your λ in TD learning, your learning method will behave differently. For $\lambda = 1$ the method behaves like a Monte-Carlo method for an undiscounted, episodic task. If on the other hand $\lambda = 0$ then the method behaves like a one-step TD method. To benefit from both λ is chosen somewhere in between those values. This can then further be applied to Sarsa. The advantage of using eligibility traces for TD learning is that they enable the application of Monte Carlo online and for continuing problems, while having computational improvements in comparison to n-step TD methods, because only one trace vector is required instead of n-feature vectors. Other advantages to step-sized methods is the continuity in the learning process and the immediate application of the learning into the behavior. Although the use of eligibility traces allows those benefits, it is still necessary to make an assumption one the optimal value function in order to estimate the value of a state, which means that even with eligibility traces the value estimate doesn't need to be correct. Accumulating trace and replacing trace are important regarding the scenario that a state gets revisited before the eligibility trace. In case of a accumulating trace, the trace will be incremented with each revisit, which can lead to traces higher than one. This doesn't happen for replacing traces, since the trace gets reset to one each revisit.

5. **1)** Forward and Backward view are equivalent regarding offline updating for $TD(\lambda)$ and $SARSA(\lambda)$, but not for online updating, though they can be similar for small values of α.
2) The value $q(s_0, a_0)$ can be estimated in forward view after taking action a_n, by first taking into account the returns of the whole trajectory like in the following formula:

$$q_{t=0}^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n R_{t+n+1} \tag{6}$$

This can then be combined to $q_{t=0}^{\lambda}$ with eligibility traces by averaging the n-step returns

$$q_{t=0}^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_{t=0}^{(n)} \tag{7}$$

So that $q(s_0, a_0)$ can be computed like this:

$$q(s_0, a_0) \leftarrow q(s_0, a_0) + \alpha(q_{t=0}^{\lambda} - q(s_0, a_0)) \tag{8}$$

In order to $q(s_0, a_0)$ before taking action a_n the returns over the trajectory have to be adjusted to the following formula:

$$q_{t=0}^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n} \tag{9}$$

Then the same steps by averaging first and then assigning the value can be taken to compute $q(s_0, a_0)$ like before. To get $q(s_0, a_0)$ in backward view the error that accumulates over the whole trajectory has to be calculated and applied to $q(s_0, a_0)$ by using the eligibility traces. Each state-action pair is assigned a error δ and eligibility trace E, which are updated each time-step and thus $q(s_0, a_0)$ is updated with each step as can be seen in the following formula:

$$q(s,a) \leftarrow q(s,a) + \alpha\delta_t E_t((s,a) \tag{10}$$

3) A dutch eligibility trace, is similar to a accumulating trace, but with smaller incremental growth.

6. Expected updates are done in order to update the values of states by getting the values of each successor state and combining that with the probabilities of occurring for each of these states. Examples for algorithms that would use expected updates are dynamic programming and exhaustive search.
   Sample updates don't involve all possible successor state, but just one sample, so one possible successor state. One example for this would be TD learning.
   Bootstrapping occurs if the update of a value involves an estimate of another value, like in dynamic programming and TD learning.
   Non-bootstrapping would be the opposite, so in order to update a value only actual observations of the environment are used, which is done in Monte Carlo.

Task II: Programming part on Eligibility Trace

1. Results of the $SARSA(\lambda)$ algorithm in SARSA_lambda.py is shown below which is provided by script itself

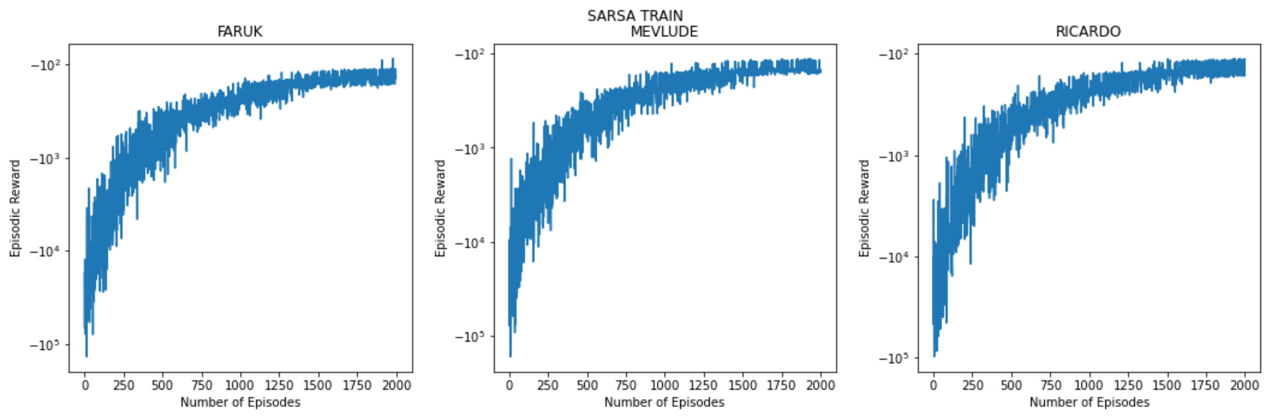


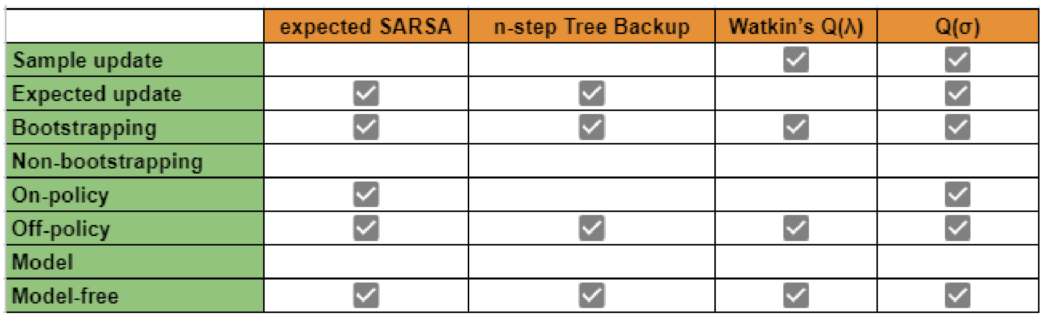
Figure 1 Results of the $SARSA(\lambda)$ algorithm for each team-member

2. Eligibility Traces includes advantages of Monte-Carlo and it can be deal with lack of Markov property. It can significantly speed learning. Bigger values of λ lead to slower convergence speed (information from the past is given a non-negligible importance). On the other hand, Q-values become lower when λ gets bigger.

   In this task, α and γ were kept constant for the experiment. The different values of λ that were used are 0.5, 0.75, 0.99. The maximum number of steps was 5000. As λ increases, the algorithm converges to a lower value than what they had with a lower λ. Interesting results were observed for Sarsa(λ) with almost λ = 1.0. (λ =0.99). This is a Monte Carlo implementation. The value function converges to the worst possible value

Bonus Tasks

1. Categorization of the given algorithms is illustrated in following figure



	expected SARSA	n-step Tree Backup	Watkin's Q(λ)	Q(σ)
Sample update			✓	✓
Expected update	✓	✓		✓
Bootstrapping	✓	✓	✓	✓
Non-bootstrapping				
On-policy	✓			✓
Off-policy	✓	✓	✓	✓
Model				
Model-free	✓	✓	✓	✓

Figure 2 Categorization of Algorithms

2. Results of the Watkin's Q(λ) algorithm in Watkins.py is shown below which is provided by script itself

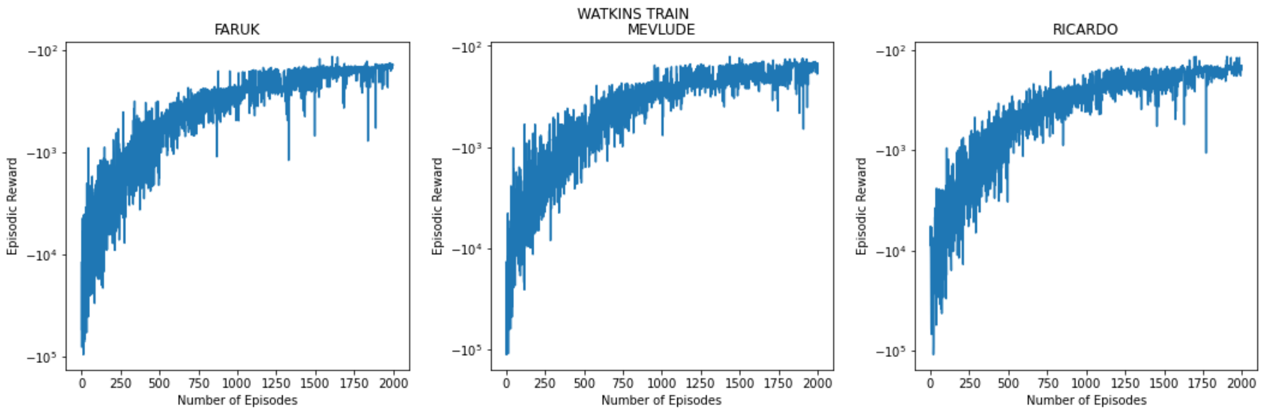


Figure 3 Results of the Watkin's Q(λ) algorithm for each team-member

3. Results of the True-online $SARSA(\lambda)$ algorithm in TO_SARSA_lambda.py is shown below which is provided by script itself

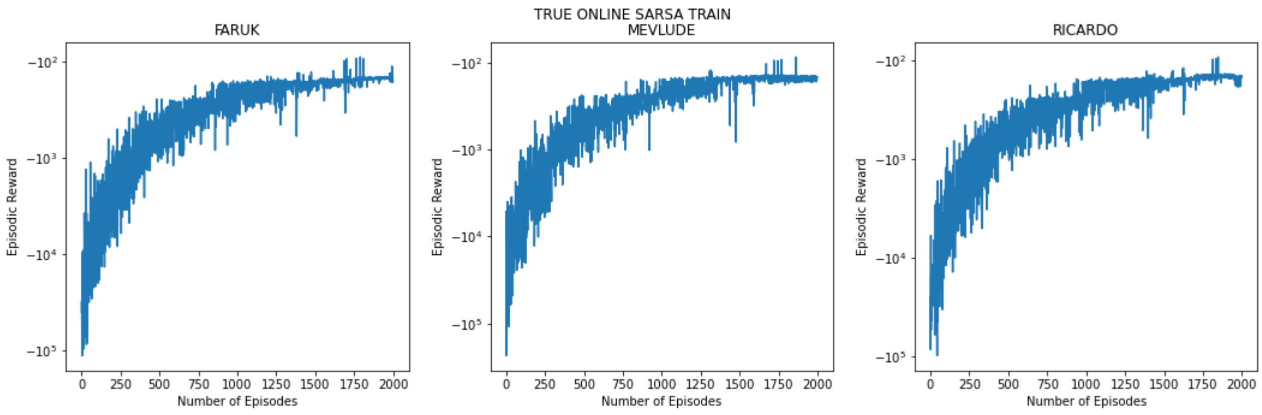


Figure 4 Results of the True-online $SARSA(\lambda)$ algorithm for each team-member

In the continuous domain, update rule is given below. This update rule applied for every step of episode

$$\mathrm{w} \leftarrow \mathrm{w} + \alpha\,(\delta + Q - Q_{\text{old}}\,)\,\mathrm{z} - \alpha\,(Q - Q_{\text{old}}\,)\,\mathrm{x} \tag{11}$$

We can think of w is equal to Q in the formula above. On the other hand, if we are in Tabular case, The update rule becomes for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ and Δ is equal to $(Q - Q_{\text{old}}\,)$ in equation 11

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\delta + \Delta)E(s,a) \tag{12}$$

And we have to update $Q(s,a)$ as described in equation 13 in each step of episode.

$$Q(s,a) \leftarrow Q(s,a) - \alpha\Delta \tag{13}$$