# ASSIGNMENT 2: MODEL-FREE REINFORCEMENT LEARNING

**Daniël Zee**
s2063131

**Noëlle Boer**
s2505169

## 1 INTRODUCTION

For this assignment we studied three model-free reinforcement learning algorithms: Q-learning, SARSA and Expected SARSA. Each algorithm was implemented in Python and our implementations were used to study their performance on a finite Markov Decision Process.
Model-free reinforcement learning is characterized by the agents learning directly from experiences with the environment as opposed to first making a model of the environment. This was also true for bandit algorithms, as the agents also learned directly from experience to estimate the action values. The algorithms studied here extend the concepts of the bandit algorithms for the use on finite Markov Decision Processes, or MDPs. A MDP is a very generic way to define sequential decision-making tasks, where actions influence not just immediate rewards, but also subsequent situations or states, and through those future rewards. We define the state space $\mathcal{S}$ as the set of states we can be in and $\mathcal{A}$ as the set of actions that are possible. Taking an action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ will bring us in a new state $s'$ according to the transition function $T(s'|s, a)$. We also receive a reward based on the reward function $r(s, a, s')$, which tells us how desirable this action was. The agent starts in some initial state and keeps making moves and collecting rewards until it has reached a predefined end state. Such a series of actions is called an episode. We have to define a policy $\pi$ for which we are trying to learn the expected rewards by keeping track of the received rewards when performing actions. In the bandit case $Q(a)$ was used to keep track of the estimated action value for taking action $a$. Because we now also have states to keep in mind we have to extend this to state-action values $Q(s, a)$. Updating the state-action values is less straightforward than the action values, as performing an action now brings us in a new state where the same action can lead to different rewards. Our state-action values should therefore not only reflect the immediate reward but also the expected reward for being in $s'$ afterwards. All three algorithms tackle this challenge by using temporal-difference (TD) learning. TD methods update $Q$ in part based on the next states current $Q$ values. $Q(s, a)$ can therefore immediately be updated after taking action $a$, without having to wait for the final outcome of the episode. We call this process of updating our estimates based on other estimates bootstrapping. For this assignment we will only look one step ahead to update our $Q$ values. This is called TD(0), or one-step TD and follows the following general steps:

1. Provide the policy $\pi$ to be evaluated
2. Initialise $Q(s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$
3. Initialise the starting state $s$
4. Loop for each step of the episode
   4.1. Sample an action $a$ from $\pi(s, a)$
   4.2. Observe the reward $R$ and next state $s'$ after taking action $a$ in state $s$
   4.3. Update $Q(s, a)$ based on $R$ combined with the attractiveness of being in $s'$
   4.4. $s'$ becomes the new working state $s$.

where step 4.3 is the main thing that differentiates the three algorithms we studied.

## 2 METHODOLOGY

We were provided with a custom environment, called the ShortCut environment, on which we studied our algorithm implementations. The environment consists of a 12x12 grid, where the agent can move to adjacent squares. The agent stays in the same square when it tries to move

outside the grid. Figure 1 shows a visualisation of the environment. The blue squares are the starting points, each with equal probability and the green square is where the agent has to get to to end the episode. The red squares are called cliffs. When the agent takes an action that ends in a cliff, it receives a reward of -100 and is returned to the starting point. All other actions return a reward of -1. The goal is to maximise the cumulative reward of each episode. The agents for each algorithm were implemented in an class containing three methods corresponding to the general TD(0) algorithm steps for which the agent is responsible: The initialisation of the $Q$ values (step 2), defining the policy function and using it to sample actions given a state (steps 1 and 4.1) and updating the $Q$ values (step 4.3). To experiment on these agents, we wrote a function for every agent that repeatedly tests the agent on an instance of the ShortCut environment. These experiment functions implement all the remaining algorithm steps which are independent of the used agent. The function runs the algorithm for a specified number of repetitions and episodes and afterwards returns the average learned $Q$ values over all repetitions and the average cumulative reward of each episode over all repetitions. These results were then used in two ways to analyze the performance of the algorithms. We use the learned $Q$ values to make plots of the action with the maximum value for each state, to observe which path a greedy policy would take for our starting positions. These plot look similar to the visualisation in figure 1. Every state the agent has been in at least once has an arrow indicating the move the greedy policy would make from there. The path the agent would take following the arrow from the initial states is highlighted in a lighter shade of green. A state with a cross indicates that all $Q$ values for that state were 0, indicating that the agent has never successfully reached this state. The red states mean something different than in figure 1, where they denote the cliffs. These are the states for which at least one of the $Q$ values was 0, indicating that the agent has not fully explored all the actions from that state. We use the cumulative rewards of each episode to plot the learning curves of each algorithm using different settings for the learning rate $\alpha$. The learning curves were smoothed with a smoothing window of size 31.

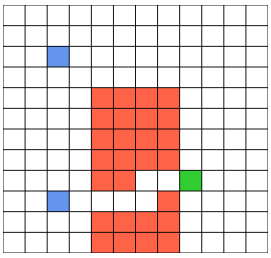

Figure 1: Visualisation of the ShortCut environment

## 3 Q-LEARNING

The first algorithm we studied is called Q-Learing. The first step of a TD(0) algorithm states that we need to provide a policy $\pi$ for which we want to evaluate the $Q$ values. Could we then just provide a greedy policy and expect the learned state-action values to be the optimal state-value function $q_*$? Why this would not work intuitively makes sense, as applying a greedy policy while learning the $Q$ values leads to very limited exploration as the agent will never take seemingly sub-optimal actions. The relation between the two goes the other way around, applying a greedy policy on $q_*$ gives you the optimal policy $\pi_*$. Q-Learning provides a way to directly learn an approximation of $q_*$ while acting on a policy that introduces more exploration. The policy we are acting on during our learning is therefore not the same policy we are learning the $Q$ values for. This is called off-policy learning. In our implementation we use a $\epsilon$-greedy policy for action selection while learning $q_*$.

### 3.1 METHODOLOGY

For the Q-Learning agent we implemented the methods for the agent class as follows:

- **Initialisation of the state-action values:** All values $Q(s, a)$ are initialised to 0.
- **Policy for action selection:** We implemented the following $\epsilon$-greedy policy:

$$\pi_{\epsilon-greedy}(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_{b \in \mathcal{A}} Q(s,b) \\ \frac{\epsilon}{(|\mathcal{A}|-1)}, & \text{otherwise} \end{cases} \tag{1}$$

- **Update state-action values:** We implemented the Q-Learning update equation to update $Q(s, a)$ values after each action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_a Q(s',a) - Q(s,a)] \tag{2}$$

where $\alpha$ is the learning rate and $\gamma$ is the discount factor.

We first ran our experiment function for this algorithm using for a single repetition of 10000 episodes using $\alpha = 0.1$ to create the greedy paths plot. Afterwards we ran the function again but for 100 repetitions of 1000 episodes using the following values for $\alpha : [0.01, 0.1, 0.5, 0.9]$ to create the learning curve plot. $\epsilon = 0.1$ and $\gamma = 1.0$ were used for all experiments.

### 3.2 Results

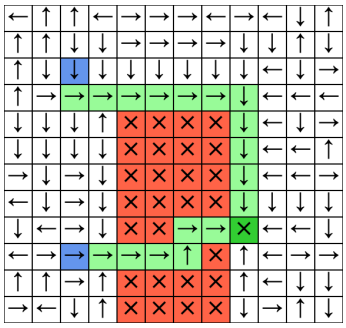

Figure 2: Paths the greedy policy would take after running Q-Learning for 10000 episodes

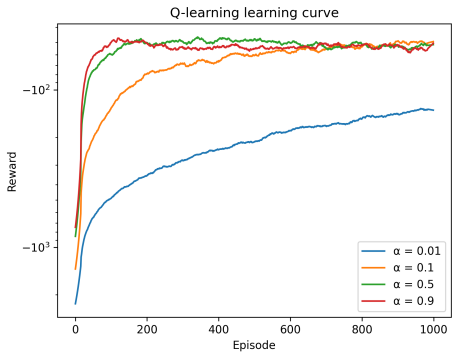


Figure 3: Average performance of Q-Learning over 100 repetitions of 1000 episodes

Figure 2 shows that the greedy policy over our $Q$ values succeeds in finding the shortest paths from the starting positions to the goal. Even the lower starting point has found the optimal path through the cliffs. This implies that we have found the optimal policy $\pi_*$. The fact that the algorithm is not hesitant to be close to the cliffs can be explained by equation 2. Our update function uses the most optimal state-action value in $s'$ for updating $Q(s, a)$, i.e. the greedy policy which we are trying to learn. When our agent is in a state next to a cliff and the $\epsilon$-greedy action selection policy decides to send us of a cliff, the $Q$ value for that action in this state will take a big hit. But as long as the other actions in that state still have a higher $Q$ value, the update function will never use this low value for calculating the $Q$ value for the states that lead to this state. So the agent is not afraid to return there as long as there is still a chance a optimal path might exist there. Figure 3 shows that a higher learning rate $\alpha$ leads to earlier convergence to the optimal average reward, which lies around -50. This might seem very low looking at the optimal paths we have found in figure 2, but we need to keep in mind that the learning curves show the rewards of the actions the $\epsilon$-greedy policy has actually taken while learning $q_*$, not the rewards of the actions $\pi_*$ would take. So even when we have converged to $q_*$ will the $\epsilon$-greedy policy still occasionally send us of a cliff, leading to lower cumulative rewards. The learning rate $\alpha$ determines to what extent the new estimation of $Q(s, a)$ should override the old value. Setting it to 0 disables any learning while setting it to 1 completely overwrites the old $Q$ value. The reason that a higher $\alpha$ always seems to give better performance is because our environment is purely deterministic and all possible randomness can therefore only come from the learned policy. But since we are learning the optimal greedy policy, where there is no randomness involved, the learning safely be set to 1 without suffering any degradation in performance.

## 4 SARSA

The next algorithm we studied is called SARSA. This is an on-policy learning algorithm, meaning that the policy $\pi$ we are trying to learn the $Q$ values for is the same policy we are using for action selecting when creating our episodes. We again used an $\epsilon$-greedy policy in order to balance exploration and exploitation. For SARSA we update the $Q(s, a)$ by combining the immediate reward $R$ with the state-action value of the action $a'$ we are taking in the next state $s'$. This means that we first have to sample $a'$ from $\pi$ before we can update $Q(s, a)$. Our general algorithm steps previously describes should therefore be altered slightly to include the sampling of $a'$ and setting $a'$ to be the new working action $a$ after each step of the episode. The algorithm is named after the fact that we need the quintuple $(s, a, R, s', a')$ to perform the update function.

### 4.1 METHODOLOGY

For the SARSA agent we implemented the methods for the agent class as follows:

- **Initialisation of the state-action values:** All values $Q(s, a)$ are initialised to 0.
- **Policy for action selection:** We implemented same $\epsilon$-greedy policy defined in equation 1.
- **Update state-action values:** We implemented the SARSA update equation to update $Q(s, a)$ values after each action:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma Q(s', a') - Q(s, a)] \tag{3}$$

We ran the same experiments for this algorithm as for the Q-Learning algorithm using the same settings to create the greedy paths and learning curve plots.

### 4.2 RESULTS

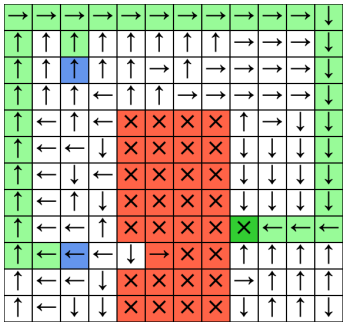

Figure 4: Paths the greedy policy would take after running SARSA for 10000 episodes

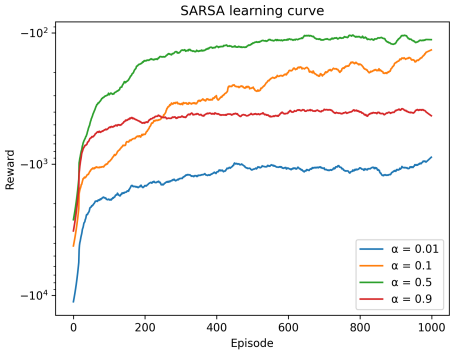


Figure 5: Average performance of SARSA over 100 repetitions of 1000 episodes

Figure 4 shows that the greedy policy over our $Q$ values results in a much longer path compared to Q-Learning, but does in the end still reach the goal. This is because these $Q$ values represent the $\epsilon$-greedy policy, meaning that it takes into account that the agent always has a chance to walk of a cliff when it is in a state close to one and therefore promotes the agent to take a roundabout path far away from the cliffs to minimize this from happening. The penalty of the small chance of walking of a cliff apparently outweighs the benefits of taking a shorter path. We see that half of the path going through the cliffs remains unexplored because of this. Figure 5 shows that a higher learning rate $\alpha$ does not automatically yield a higher cumulative reward after all episodes. This is because the policy we are using for the update function is probabilistic, which means that we have randomness involved in updating the $Q$ values. Putting too much emphasis on recent observations would not average out this randomness in the $Q$ values. We should therefore not set our learning rate too high, as that would make the $Q$ values diverge over many episodes. Figure 6.3 on page 133 of Sutton & Barto (2018) shows this effect nicely and suggests that for $\alpha = 0.1$ we would eventually converge in the mean over many repetitions to the optimal cumulative reward if ran on more episodes than we did here. The figure on page 132 of Sutton & Barto (2018) shows a higher optimal cumulative reward for SARSA compared to Q-Learning. We do not see the same results because none of our settings reached the optimal cumulative reward.

## 5 STORMY WEATHER

We tested the Q-Learnig and SARSA algorithms in a modified version of the ShortCut environment, called the windy ShortCut environment. After each action there now is a 50% chance that the agent is moved to the square below. This results in our environment changing from a deterministic to a probabilistic one. We again ran our algorithms for a single repetition of 10000 episodes using $\alpha = 0.1$ to create the greedy paths plot.

### 5.1 RESULTS

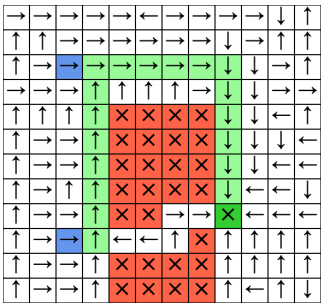

Figure 6: Paths the greedy policy would take after running Q-Learning for 10000 episodes on the windy ShortCut environment

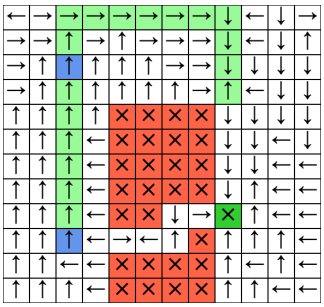

Figure 7: Paths the greedy policy would take after running SARSA for 10000 episodes on the windy ShortCut environment

Figure 6 shows that Q-Learning no longer takes the shortest path through the cliffs from the lower starting position. This is because there is now a big risk of falling of a cliff when taking this path. Q-Learning takes this stochastic wind into account when approximating $q_*$ and the resulting $\pi_*$ as a result prevents the agent from being in a state where the downwards wind would send it of a cliff. Because the environment has become probabilistic is also important to note that $\alpha$ cannot be set to a high value without degrading long term performance. The path SARSA takes from the starting positions also avoids the states close to the cliffs, but this is not different from the behaviour we observed in the regular environment as we were already taking into account a random chance of falling of a cliff. Following the path for SARSA seems to end in an inescapable loop where we try to go up and down indefinitely. While it might look that way, we need to keep in mind that the agent always has a chance to be blown down one square and continue its path to the goal. If we were to adjust the direction of the wind to be upwards, we would most likely observe the same effects, as the chance of falling of the upper cliffs in the shortest path is just as likely. Changing the wind direction to be sideways disincentives taking the shortest path less as the path is mostly horizontal.

## 6 EXPECTED SARSA

The final algorithm we studied is called Expected SARSA. Expected SARA is an on-policy learning algorithm like SARSA, but instead of updating $Q(s, a)$ based in part on $Q(s', a')$, which involves sampling $a'$ from our probabilistic policy and therefore introducing randomness to our learning, we instead update $Q(s, a)$ based on the expected $Q$ over all actions in $s'$ under the current policy. We therefore do not need to know $a'$ when updating $Q(s, a)$, making our update function fully deterministic like Q-Learning. Q-Learning can be seen as an off-policy version of Expected SARSA using the probability distribution of the greedy policy to calculate the expected Q value in $s'$.

### 6.1 METHODOLOGY

For the Expected SARSA agent we implemented the methods for the agent class as follows:

- **Initialisation of the state-action values:** All values $Q(s, a)$ are initialised to 0.
- **Policy for action selection:** We implemented same $\epsilon$-greedy policy defined in equation 1.
- **Update state-action values:** We implemented the Expected SARSA update equation to update $Q(s, a)$ values after each action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \sum_{a} \pi(a|s')Q(s',a) - Q(s,a)] \tag{4}$$

We ran the same experiments for this algorithm as for the Q-Learning and SARSA algorithms using the same settings to create the greedy paths and learning curve plots.

### 6.2 RESULTS

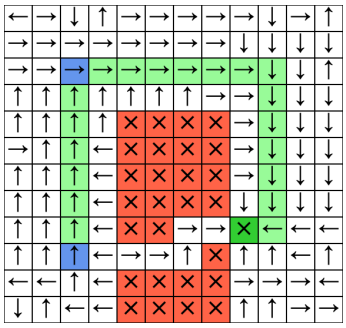

Figure 8: Paths the greedy policy would take after running Expected SARSA for 10000 episodes

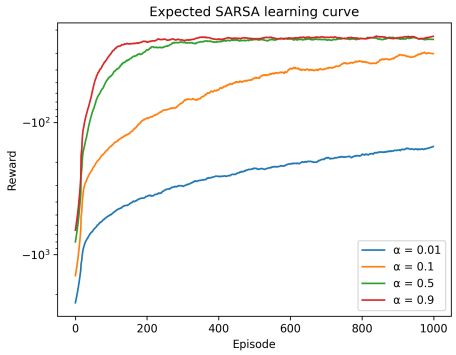


Figure 9: Average performance of Expected SARSA over 100 repetitions of 1000 episodes

Figure 8 shows that the greedy policy over our $Q$ values finds a longer path to the goal, leaving always one square between the agent and the cliff. This is to be expected because just like with SARSA we are acting on the $Q$ values representing a safe path for the $\epsilon$-greedy policy. The question then arises why the path is not the same as for SARSA in figure 4. This is because we only run a single repetition and for $\alpha = 0.1$ we know that SARSA only converges to the true state-action function $q_\pi$ in the mean over many repetitions. Because Expected SARSA is deterministic like Q-Learning and we are working with an deterministic environment, Expected SARSA and Q-Learning are guaranteed to find $q_\pi$ (which is the same as $q_*$ for Q-Learning) in every repetition given enough episodes. We can therefore conclude that figure 4 shows the optimal path for the $\epsilon$-greedy policy. Figure 9 confirms our deduction that $\alpha$ can again safely be set to 1 without suffering any degradation in performance. The optimal average reward to which the algorithm converges lies around -25, which is higher than the value found using Q-Learning in 3. This is because we are acting on the same policy we are learning, leading to optimal cumulative rewards for that policy after convergence.

## 7 COMPARISON AND CONCLUSION

Between the three studied algorithms we have found two key aspects that differentiate the observed behaviour on our environment: whether the algorithm is off-policy or on-policy and whether the update mechanism is deterministic or probabilistic and the effects this has on the learning rate $\alpha$. Q-Learning, an off-policy algorithm, has the advantage of directly approximating $q_*$, independent of the policy being followed to select the actions during learning. A disadvantage we have seen is that the average cumulative reward does not reflect $\pi_*$, which we are trying to learn. This could however be solved by gradually decreasing $\epsilon$ during the learning process as this would gradually move it to be an on-policy algorithm and make the agent always act optimally. SARSA and Expected SARSA, both on-policy algorithms from the start, do not have this disadvantage. These algorithms have the disadvantage of never finding the shortest path through the cliffs in the environment because we are learning the $Q$ values for a sub optimal policy in order to have enough exploration during learning. We have found that the presence of randomness in either the environment or the update mechanism of the chosen algorithm is the main factor in choosing the learning rate $\alpha$. When both are completely deterministic we do not need to average over our estimated $Q$ values so $\alpha = 1$ will always be optimal. When the environment or the update mechanism introduces randomness, like in the windy environment or when using the SARSA algorithm, do we need to set our $\alpha$ sufficiently low as to not diverge in the long run. Another option which we have not explored for this assignment is to gradually lower $\alpha$ over time like in the incremental mean update, where we use $\alpha = \frac{1}{n}$. This approach is guarantied to converge to $q_\pi$ for every repetition given enough episodes according to page 124 in Sutton & Barto (2018). Further research could therefore be done using decreasing values of $\epsilon$ and $\alpha$. We could also try experimenting with different values for the discount factor $\gamma$.

## REFERENCES

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.