Reinforcement Learning: Project Report

03-04-2025

By Ella Kemperman, student number *s1119890*, Ioana-Anastasia Orasanu, student number *s1112418*, Rostyslav Redchyts, student number *s1113227*, Mila van Rijsbergen student number *s1128515*.

1. Introduction

Reinforcement Learning (RL) has become increasingly relevant in Machine Learning over the past decade. RL has been irreplaceable in robotics for a while, and recently, it has also contributed to the AI revolution through approaches like RLHF (Reinforcement Learning with Human Feedback). Despite its prevalence, RL is already a mature and complex field, which can be intimidating to newcomers. Hence, in this report, we look over a few simple algorithms to explain how they work and get some insight into how they differ from each other.

More specifically, various RL algorithms were implemented, and their performance in a certain environment was compared. This environment was designed specifically for this project and involved an agent, also referred to as 'the cat', navigating a 2-dimensional grid while chasing a target, metaphorically referred to as 'the mouse'.

The environment (see Figure 1) is discrete, with both the agent and the mouse traversing it by making a step in one of four directions. To increase the complexity of the environment, two obstacles have been added. Firstly, special cell types were included, called 'trees', where the cat can enter, but the mouse cannot. Secondly, walls in between the cells were added, impermeable for the cat, while the mouse can go through them.

The agent receives a penalty of -1 for each step that resulted in an empty cell, a reduced penalty of -0.1 for landing on a tree, and a reward of +20 for catching the mouse (regardless of the cell type). These rewards were chosen to motivate the agent to catch the mouse as quickly while also giving some space for developing halting strategies.

The project consists of comparing two algorithms from Dynamic Programming (DP), which include Policy Iteration and Value Iteration (discussed in Section 2; a Monte-Carlo method (discussed in Section 3); and three Temporal Difference (TD) algorithms (mainly discussed in Section 4): Sarsa, Q-Learning, and Deep Q-Learning (discussed in Appendix A). To achieve this, the performance of the learned policies was compared as they were being refined during learning. Furthermore, the final policies and their returns are compared as well. This is done in Section 5.

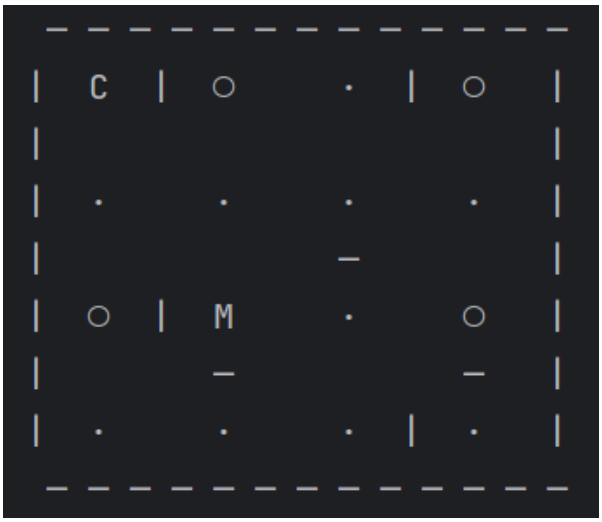


Figure 1: A visual representation of the grid environment. The '|'s indicate walls, the 'o's indicate trees, 'C' is the agent, and 'M' is the mouse.

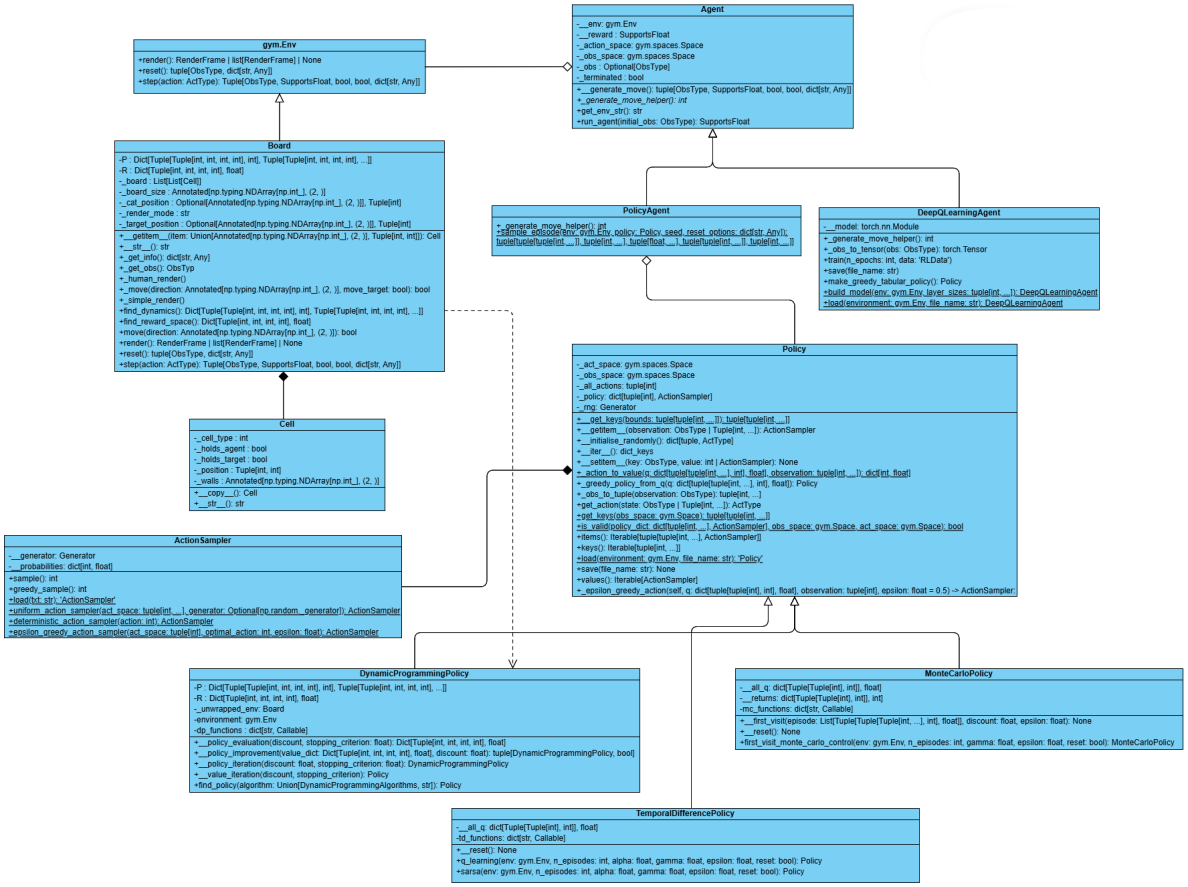


Figure 2: Class diagram of the implementation, made with Visual Paradigm. Note that the alias `gym` was used for the gymnasium package and the alias `np` was used for the numpy package.

2. Dynamic Programming

The idea behind the first family of algorithms discussed — Dynamic Programming — lies in the approach of dividing a large problem (e.g. finding an optimal policy) into smaller sub-problems, like improving a current policy, based on the current estimate of the value function. Solving these smaller subproblems is guaranteed to lead to an optimal solution to the main problem (although it might take infinite iterations).

For this project, two DP algorithms were implemented and tested: Policy iteration and Value Iteration.

2.1. Policy Iteration

The main idea of Policy Iteration is to find an optimal solution through planning. Specifically, the process consists of iteratively evaluating the current policy (through Policy Evaluation) and improving this policy, based on the updates estimate of the value function (through Policy Improvement), until convergence.

Refer to Figure 3 for a full pseudocode.

Policy Evaluation

The policy evaluation algorithm relies on the Bellman expectation equations to iteratively update the estimates for the value of each state by considering the values and the rewards of the successor states, weighted by the probability of reaching that state. By iteratively updating the value estimates for each state, the optimal value function can be approximated, omitting the need for complex analytical solutions.

Note that the convergence to the optimal value function, although guaranteed over infinite iterations, is likely to be computationally expensive. To avoid this issue, a variable delta is defined as the maximal value difference through all the states after the current iteration. When this value becomes low enough, it signifies that the current estimate is close to the true value, and so the algorithm is terminated.

Policy Improvement

Policy Improvement algorithm relies on the estimated value function, obtained through Policy Evaluation to determine a new, improved policy though greedily choosing the most promising action. In other words, for every state, the action that is expected to lead to the highest overall reward is chosen for the updated policy.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 3: Policy Iteration pseudocode (Sutton & Barto, 2018, p. 80)

Implementation

This algorithm was implemented as a set of methods in the `DynamicProgrammingPolicy` class. This allows the algorithm to update the policy by just modifying itself. The policy itself is stored as a dictionary, mapping each possible observation to the action that is taken. The state-value function is also represented as a dictionary that maps state to value. The policy iteration and policy improvement parts were separated into two methods, the former calling policy evaluation and then policy improvement, until no more changes to the policy are made.

2.2. Value Iteration

The principle of optimality states that a policy π achieves the optimal value from state s, $v_\pi(s) = v^*(s)$ if and only if from any state s' reachable from s, π achieves the optimal value $v_\pi(s') = v^*(s')$. Thus, this theorem directly leads to the value iteration algorithm. With value iteration, the most optimal state value function is computed by iteratively updating the estimate. The algorithm outputs a new approximately optimal policy.

The process starts by selecting a random value function, over which is iterated using the Bellman equations as an update rule, updating the values for all states. This process gets repeated until convergence to optimal value function, which is guaranteed. Once the optimal value is found, it takes the actions such that the maximum in the Bellman optimality equation is satisfied. For the pseudocode, see Figure 4.

Implementation

Value iteration was implemented in code as a method of the `DynamicProgrammingPolicy` class. As can be seen in Figure 2, this class inherits from `Policy`. The algorithm mainly works by storing the state values in a dictionary that has as keys the states and as value the state value. It updates this dictionary using the value obtained by using the Bellman equation on each possible action, picking the value where the action is maximal. It then also updates the action taken in the policy to the new best action.

Convergence is determined by actively checking the value of *delta* at all states. Once it becomes small enough, the process ends. Since the action taken in each state was already updated during the running of the algorithm, the policy is now approximately optimal.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 4: Policy Iteration pseudocode (Sutton & Barto, 2018, p. 80)

3. Monte-Carlo

Monte Carlo (MC) updates the model using data obtained from running simulations of the environment. This allows it to work without assumptions about how the model should work. How MC updates its values differs per method used. In this report, the MC First-Visit on-policy algorithm was used. The pseudocode is given in Figure 5.

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary ε-soft policy
  $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
      $A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Figure 5: MC on-policy first-visit pseudocode (Sutton & Barto, 2018, p. 101)

3.1. First-Visit

After each episode, with First-Visit, the program takes the information learned in an episode and assigns for each state-action pair the gained reward for taking the action plus the discounted total reward for the next pair, see Equation 1.

$$q(S_t, A_t) = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) \tag{1}$$

With S_t the state at time t, A_t the action taken at time t, R_t the reward obtained at time t, γ the discount, and q the function mapping state and action to their value. This is done for all first encounters of a state-action pair in the episode. Afterwards, the average for each state-action pair is determined over all the values gained for a state-action pair over n episodes.

3.2. On-Policy

After each episode, the program looks at all the acquired state-action values. It determines the best action for each state based on which state-action pair has the highest value. To make sure all states and actions are seen by the program, the policy is updated with an ϵ-soft policy, the best action to take will have a higher chance of getting picked but still allow the other actions to be chosen as well. An epsilon soft policy means that the probability of each action given the state is given by Equation 2:

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \varepsilon/|A(s)| & \text{if} \quad a = \arg\max_a Q(s,a) \\ \varepsilon/|A(s)| & \text{if} \quad a \neq \arg\max_a Q(s,a) \end{cases} \tag{2}$$

This way of updating the policy guarantees that the new policy is better than the one before but it cannot guarantee an optimal policy.

3.3. Implementation

The algorithm was implemented as a method of the `MonteCarloPolicy` class (for class structure, see Figure 2). The code first samples an episode using itself as a policy, and then updates its internal Q function, which is a dictionary mapping state-action to a reward, based on the episode returns. The policy then updates itself by creating an ϵ-greedy `ActionSampler` on the optimal action, assigning it to the current state. Note that ϵ does not decay, so the policy will always stay ϵ-soft.

4. Temporal Difference

Temporal Difference (TD) Learning algorithms are another set of algorithms that learn the environment from experience rather than by computing the optimal policy based on a model. It achieves this by learning the optimal state-action value function $q^*(s, a)$ by sampling many episodes. While this is similar to Monte-Carlo methods described in Section 3, Temporal Difference learning does not use the full return computed at the end of the episode; rather it uses bootstrapping to allow it to update the $Q(s, a)$ after each step taken in the episode. This enables it to be used in environments without terminal states. In comparison to Monte-Carlo, TD algorithms generally have higher bias but lower variance.

Two main variants of Temporal Difference learning are discussed in this report: SARSA, an on-policy control algorithm, discussed in Section 4.1, and Q-learning, an off-policy control algorithm, discussed in Section 4.2.

4.1. SARSA

SARSA is an on-policy temporal difference learning control algorithm. SARSA is short for 'State, Action, Reward, State, Action', which refers to how the state-action value function is updated at each time step. The goal of the algorithm is to approximate $Q(s, a) \approx q^*(s, a)$. This is done by sampling many episodes, and for each step of the episode, updating the value function using the following update rule:

$$Q(S, A) := Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)) \tag{3}$$

Note that α is the learning rate, γ is the discount factor, S is the current state, S' is the next state, which is reached after taking action A. Finally, A' is the action taken from state S'. This action is generated by acting ϵ-greedy on Q for the given state.

The algorithm works by first initalising for all state, action pairs, $Q(s, a) = 0$. It will then repeat the

following over a prespecified number of episodes. It first picks an action taken from the starting state of the episode, after which it will first take the determined action, get the new state S', use this new state together by acting ϵ-greedy on the value function to get a new action A', after which Equation 4 is used to update the state-action value function. This is repeated until the episode is finished.

Compared to Q-Learning, SARSA generally chooses a safer path during training, which makes it more likely to converge on an optimal policy, while it does cause it to need more steps than Q-Learning.

Implementation

This algorithm was implemented as a method of the `TemporalDifferencePolicy` class (for class structure, see Figure 2). The code learns over n episodes, each episode deciding its actions ϵ-greedy, updating its state-action value function Q along the way, which is represented by a dictionary mapping state-action pairs to their value. When finished, it computes a new policy by acting greedily on the value function. This policy is then returned. Note that usually, the policy derived from Q is ϵ-greedy. However, this implementation acts greedily instead in the final policy, since it is assumed to have arrived at an approximately optimal policy.

4.2. Q-Learning

Q-learning is an off-policy algorithm, meaning that it learns the value of the optimal policy independently of the action values ("Artificial Intelligence - foundations of computational agents – 11.3.6 On-Policy Learning", n.d.). The algorithm is very similar to the value iteration one. While value iteration tries to find the optimal value, q-learning aim at estimating the optimal Q-function. Same as for value iteration, convergence is guaranteed for sufficient experience. The goal of Q-learning is to determine the optimal action based on the current state. The update rule of Q-learning is as follows:

$$Q(S, A) := Q(S, A) + \alpha(R + \gamma \max_a Q(S', A') - Q(S, A)) \tag{4}$$

The algorithm works by initializing a Q-table, which stores the action-values sets. The process continues by selecting and performing an action, after which the reward is measured and the the Q-table gets updated. The process ends when a terminal state *s* is reached.

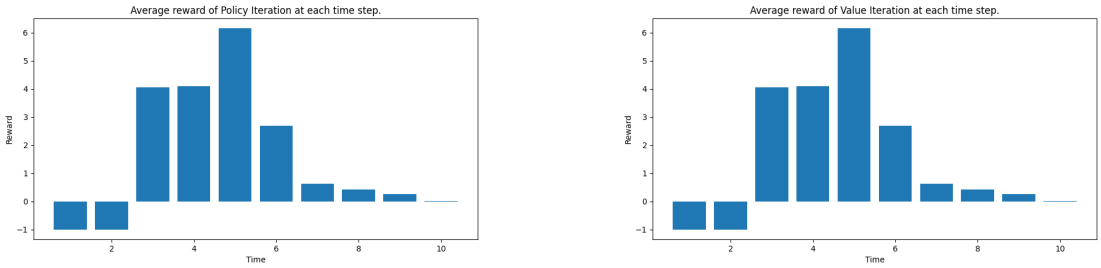
Of great importance is that Q-learning has maximization bias. A solution to this is Double Q-learning, which uses two Q-functions but does not get updated at the same time.

Implementation

The implementation of Q-Learning is extremely similar to that of SARSA. The main difference in the implementation is that the action chosen to update the Q dictionary is a greedy action on Q, instead of the actual action taken. Other than that, there are no differences with SARSA.

5. Comparison and Discussion

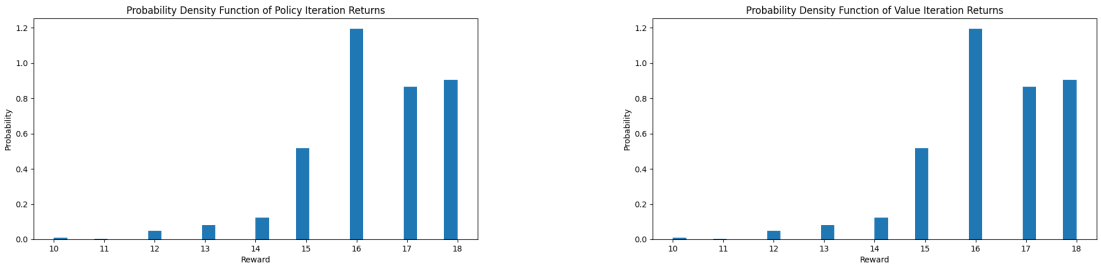
As can be seen from the plots below, the results derived from Policy Iteration and Value Iteration are strikingly similar. This indicates that both algorithms likely converged to an optimal policy. Both policies seem to mostly reach the terminal state after taking 3-6 steps with the highest average reward being at the fifth time stamp. This indicates that most episodes end quickly, suggesting a great performance of the determined policy.



(a) Policy Iteration **(b)** Value Iteration

Figure 6: Average reward at each time step for both Policy Iteration and Value Iteration.

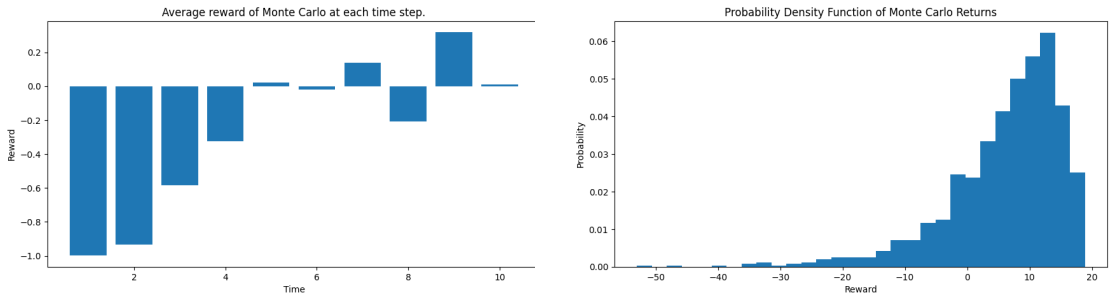
This interpretation is further supported by the plots of the probability density functions of the returns derived from both policies. The plots are once again (almost) identical and indicate that the highest reward likelihood for these policies is around 16, aligning with the 3-6 steps with the rewards of -1 from before.



(a) Policy Iteration **(b)** Value Iteration

Figure 7: Return probability distributions for Policy Iteration and Value Iteration.

The Monte Carlo algorithm, on the other hand, failed to arrive at an optimal policy over 100 steps (more steps did not help, as can be seen in Figure 10). The plots from Figure 8 indicate a lower average reward for all time steps, compared to the DP algorithms from earlier. Furthermore, the probability density function suggests that the policy, derived from First Visit Epsilon Greedy Monte Carlo, results in a much higher likelihood of returning a negative reward than any of the DP policies. This can be explained by the fact that Dynamic Programming algorithms have a full context of the environment and use planning internally, while Monte Carlo needs to rely on the limited information obtained through the interaction with the environment.

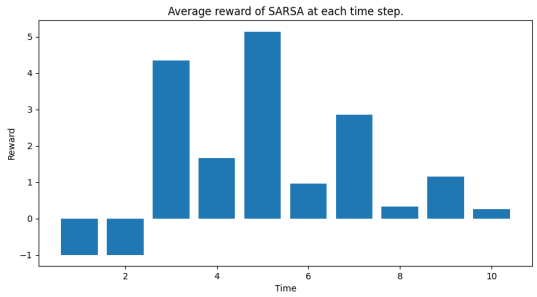


(a) Average reward at each time step **(b)** Return probability distributions

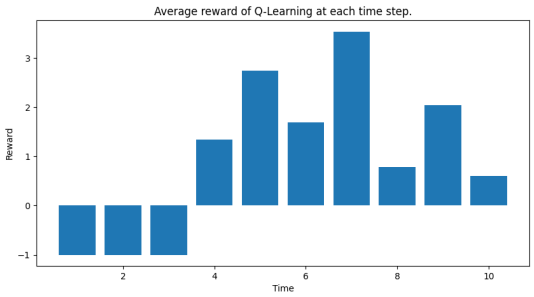
Figure 8: Average reward over time and the Return probability distribution for Monte Carlo.

As for the Temporal Difference algorithms (see Figure 9), both SARSA and Q-Learning seem to perform quite similarly, with SARSA displaying a slightly better performance. Similar to results from the DP algorithms, it can be seen that the first two steps yield a negative reward, though unlike Q-Learning,

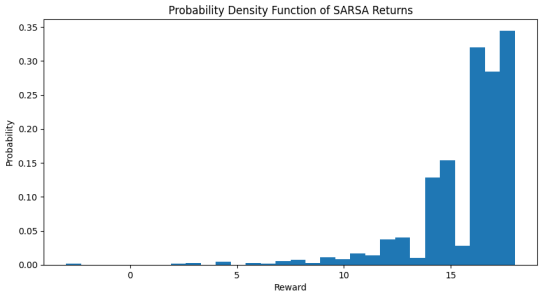
SARSA learned to obtain significant positive reward on the third step. The return probability distributions indicate a low chance of receiving a negative total reward, displaying a significantly higher performance compared to MC.



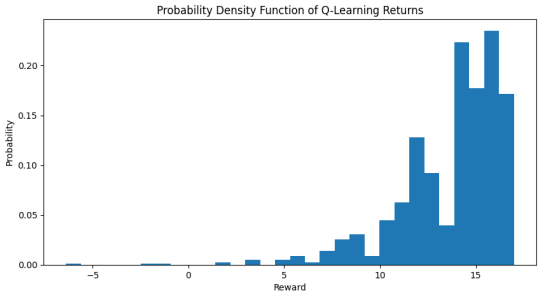
(a) SARSA: Average reward at each time step



(b) Q-Learning: Average reward at each time step



(c) SARSA: Return probability distribution



(d) Q-Learning: Return probability distribution

Figure 9: Comparison of SARSA and Q-Learning. Top row (a, b): Average reward over time. Bottom row (c, d): Return probability distributions.

When comparing the learning behaviour of First-Visit Monte Carlo, Q-Learning, and SARSA (see Figure 10), it can be seen that both SARSA and Q-Learning stabilise quite quickly at the same average reward obtained, after about 50 learning steps. Monte-Carlo also stabilises at this point, although it is less stable and has a lower average value.

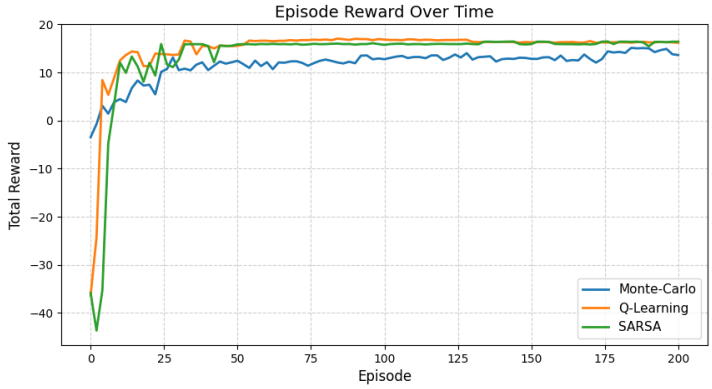


Figure 10: Comparison of the average reward evolution over the course of learning of First-Visit Monte Carlo, Q-Learning, and SARSA

6. Conclusion

In this report, some common algorithms used in Reinforcement Learning (RL) were discussed. These algorithms include Policy Iteration, Value Iteration, First-Visit Epsilon-Greedy Monte Carlo, SARSA, and

Q-learning. Having analysed data from all the algorithms (see Figure 11), it was found that Dynamic Programming algorithms (Policy Iteration and Value Iteration) arrived at the best-performing optimal policy. This result indicates that for the cases where the environment is known (or can be easily learned), DP is a great choice.

SARSA and Q-Learning fall shortly behind, with SARSA slightly outperforming Q-Learning. Those algorithms did not have any direct knowledge of the environment, and yet were able to arrive at a great policy after seeing 100 episodes for the environment used in this report. Q-Learning 'prefers' to act more risky, resulting in a lower average reward.

First-Visit Epsilon-Greedy Monte Carlo yielded the worst results of all the policies. The results clearly indicate that it performs significantly better than the random policy, indicating that it did learn from the environment. Yet, First Visit MC only learns from the first instance of the state-action pair in an episode, making the convergence slower.

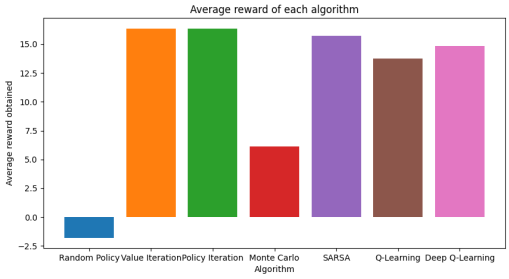


Figure 11: Comparison of the average (over 1000 episodes) final return achieved by each algorithm.

References

Artificial Intelligence - foundations of computational agents – 11.3.6 On-Policy Learning. (n.d.). https://artint.info/html1e/ArtInt_268.html

Qin, Y. (2025). Tabular Learning and Beyond. https://brightspace.ru.nl/d2l/le/content/502312/viewContent/3008418/View

Sutton, R. S., & Barto, A. G. (2018, November). *Reinforcement Learning, second edition*. MIT Press.

A. Deep Q-Learning

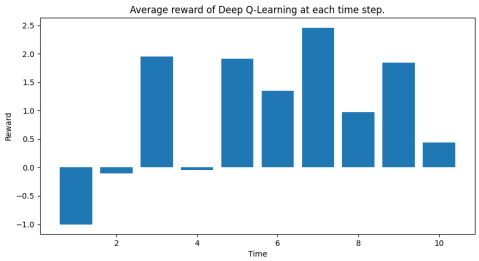
As an additional exercise, Deep Q-learning (DQL) was implemented. In contrast to the algorithms discussed in the main report, Deep Q-Learning is not tabular, meaning that it does not compute a lookup table for every single state. This is mainly useful when dealing with environments that have an extremely large observation space since it would be impossible to use tabular methods for these types of environments since training would take too long while likely also taking too much space to store all information.

A.1. Theoretical Deep Q-Learning

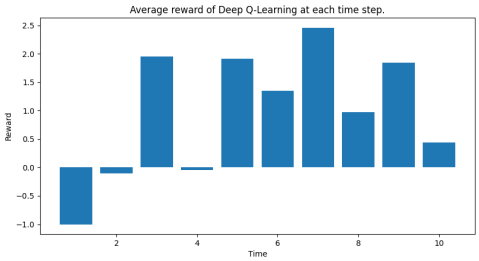
Deep Q-Learning uses a neural network to circumvent the need for lookup tables. At the output layer, this neural network needs to have one neuron for each possible action. The output value of a single neuron in the output layer represents the value of this action in the given state. From these output values, the optimal action can for instance be selected greedily by picking the action with the largest value (softmax could also be used, picking each action at the probability specified by softmax).

The input layer of the neural network needs to be able to take an observation from the environment. In the case of the environment used in this report, there are 4 input neurons, the first two representing the agent position, the latter two representing the target position. However, for an agent that would for example observe the world with a camera, the flattened image could be used as input for the network.

In order to train a neural network, gradient descent (GD) is used. To allow GD to be used, there needs



(a) The average reward received by Deep Q-Learning at each time step.



(b) Return probability distributions

Figure 12: Average reward over time and the Return probability distribution for Deep Q-Learning

to be a loss function. According to Qin (2025), the loss for DQL can be computed using Equation 5:

$$L_t(\mathbf{w}_t) = \mathbb{E}\left[\left(R_{t+1} + \gamma \max_{a'}(S_{t+1}, a'; \mathbf{w}_{t-1}) - Q(S_t, A_t; \mathbf{w}_t)\right)^2\right] \tag{5}$$

With t the time, R_t the reward at time t, S_t the state at time t, A_t the action taken at time t, $\mathbf{w}_t$ the weights of the model at time t, a' the set of actions that can be taken, and Q the model, that gives the value of a state and action computed using the weights of the model.

A.2. Practical Deep Q-Learning

The basic implementation of DQL is fairly simple. Two models are maintained, the previous model and the current one. The model is trained by looping over many episodes, at each step in the episode, obtaining the reward and next state, then taking an action using some policy, and using this information to compute the loss using Equation 5.

There are, however, some issues with this implementation of DQL. Firstly, the target is determined by the previous model. Since this model is constantly updating, the algorithm is chasing a moving target, which can lead to algorithm instability (Qin, 2025). To solve this issue, instead of moving the target after every step, the target is moved after n steps. This somewhat improves model stability.

This improvement to the base algorithm was still not enough to make DQL work. This was caused by that the model trains on each episode only once, and that all data it trains on is sequential, making the model constantly need to adapt to the current episode. To circumvent this issue, experience replay was used. Many episodes were sampled, for each episode storing a tuple of (S, A, R, S', T) (with T indicating if S' is terminal). This data was then randomly shuffled and trained on using mini-batch gradient descent over a few epochs. This allowed the model to converge on a good policy.

The final model trained was trained on 100000 randomly sampled episodes, training on it in 10 epochs. A learning rate of 0.01 was used, a discount of 0.9 was used, and the batch size was set to 1024. The model was set to reset the target model every 10 batches. The final model receives the average rewards plotted in Figure 12a at every time step when the agent starts in the top left corner of the environment and the agent starts in the bottom right. Furthermore, the probability distribution of the return was plotted in Figure 12b. Note that there were a few extremely bad episodes in there, but in general, the model performs quite well, although not as well as SARSA and Q-Learning. Furthermore, the model does take quite some time to train, and sampling all the data required takes even longer.

In conclusion, the Deep Q-Learning algorithm discussed in this report seems to be able to somewhat approximate an optimal policy, and in cases where tabular algorithms are not possible, DQL could be a good alternative. However, when tabular solutions are feasible, it seems to be more efficient to just use a tabular algorithm such as Q-Learning instead.