Reinforcement Learning Project: Dealing with sparse rewards in the Mountain Car environment

Silvia Romanato 360412 – Alexi Semiz 368603
Department of Computer Science, EPFL, Switzerland

I. Introduction

In this project, we apply Reinforcement Learning (RL) algorithms to the Mountain Car problem in the Gymnasium environment, where an underpowered car must build momentum to reach the top of a hill. The agent can accelerate left, right, or do nothing, and it receives a reward of -1 per time step until the goal is reached, presenting a challenge due to the lack of positive feedback. We implement and compare two RL algorithms: the model-free Deep Q-Network (DQN), which learns from direct interactions, and the model-based Dyna, which also incorporates simulated learning for improved efficiency. The project code is available here.

II. Random Agent

A random agent was tested for 100 episodes in a simulation, where it made random decisions at every step. The agent consistently scored -200, indicating a failure to improve, as it received a -1 reward for each of 200 steps before each episode was truncated. The episodes lasted about 0.0025 seconds on average, with slight fluctuations early on. (see the result on figure 1. Due to the agent's consistent failure, alternative algorithms will be considered.

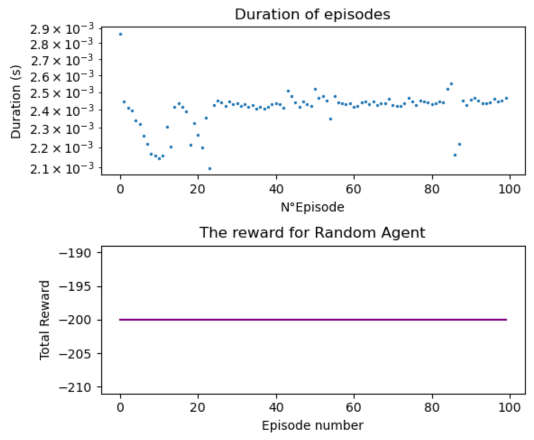

Fig. 1: Random Agent - Duration and Reward

III. DQN Agent

The next algorithm we will implement is the deep Q-network (DQN) algorithm. This is a model-free, online, off-policy reinforcement learning method. DQN combines Q-learning with deep neural networks to learn an optimal action-selection policy for an agent interacting with an environment.

A. Overview of the method

The DQN agent learns by interacting with the mountain car gymnasium environment and observing its current state and taking an action. The environment transitions to a new state and provides a reward signal based on the action taken. The agent chooses an action based on ϵgreedy policy where the ϵ is decreased over the course of learning through exploration annealing. (see the detailed overview of the method in the appendix A)

B. DQN with Sparse reward Results

In the initial implementation of the DQN agent, we train the `DQNAgent` using the standard sparse reward setup over a course of 1000 episodes. The corresponding results are displayed in Figure 2.

To manage terminal states and avoid large gradients, we ensure that gradient calculations are performed only for non-terminal (s, a, r, s') transitions within the experience replay buffer and the gradients are clipped because we use the AdamW loss. Alternatively, the Huber loss could be used without gradient clipping.

The target network in the Deep Q-Network (DQN) algorithm is not updated at every step but is instead updated every 1000 steps. Updating the target network less frequently decouples the target Q-value calculation from the Q-value being updated, reducing the harmful correlations between the two and mitigating the overestimation bias in Q-learning. By holding the target network's weights fixed for a certain period, the target Q-values remain stable, preventing the moving target problem and allowing the Q-network to converge more reliably ([1]).

In these plots the rewards, successes, cumulative reward, loss and duration per episode are displayed. What can be concluded is that no success can be achieved and that the agent does not manage to increase the reward over the 1000 episodes. The challenge in a sparse reward environment is that the agent lacks information on which actions will lead to increased rewards. This lack of guidance significantly complicates navigation within such environments. Consequently, the agent acts following random exploration, and it often runs out of time before reaching the goal due to episode truncation of the episode after 200 steps. Over time, although the loss decreases, it stops to make meaningful adjustments to the agent's network, making impossible to achieve further improvements and learning. Implementing an auxiliary reward function could enhance this situation by providing more immediate feedback,

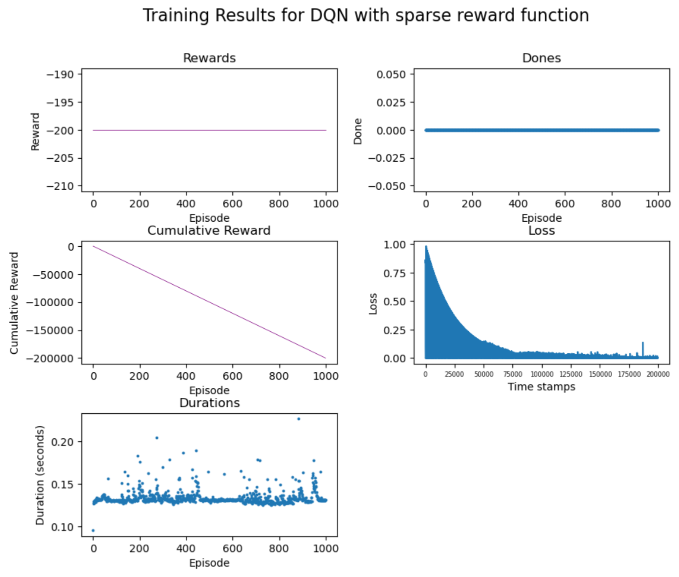

Fig. 2: Results plots for DQN with sparse rewards: total and cumulative rewards, successes, loss and duration per episode.

and therefore enabling the agent to learn about the environment more effectively.

C. DQN with heuristic reward

To direct the agent toward the goal, we have implemented a heuristic reward function. Given the environment and the car's limited acceleration, it is beneficial to encourage the agent not only to aim for the right goal hill but also to reward it for reaching the top of the left hill, as this position gives it enough kinetic energy.

$$\begin{aligned} \text{heurR} &= -c(1-\text{normPos}) - d(1-\text{height}) - \text{stepFine} \quad (1) \\ &= -c(1-x) - d(1-\cos(\pi x - \frac{\pi}{1.5} + 0.3)) - 10 \quad (2) \end{aligned}$$

In order to do so, the developed function of the heuristic reward calculation is represented in 1 and 2 and is implemented as follows. Position and height are normalized `normPos` between 0 and 1, where 1 is the goal. Then, the location of the agent is evaluated for their proximity to the goal and elevation achieved, respectively, with less negative rewards granted as the agent approaches the goal or ascends higher. Constants `C` and `D` scale the impact of these factors on the total reward. Additionally, a `stepFine` is subtracted for each action that doesn't conclude the episode, incentivizing not only the achievement of the goal but also efficiency in reaching it quickly. This composite approach ensures that the agent's training is balanced across achieving the objective, optimizing path efficiency, and managing vertical challenges. Finally, the heuristic function is multiplied by and heuristic factor `heurFactor` which accounts for the impact of the heuristic function to the total reward, and then this is summed up to the original reward. Therefore, the total reward looks as follows:

$$\text{totalR} = \text{origR} + \text{heurR} * \text{heurFactor}$$

Using this heuristic reward and setting `C` to 1, `D` to 0.5 and `stepFine` to 0.3. Additionally, the `heurFactor` is set to 1. When the reward is decreased, the environment increasingly resembles a sparse one, and as a result, the agent consistently fails to achieve the goal, as observed in the previous chapter. On the other hand, by increasing the `heurFactor` too much, it happens that it fails to succeed too, but this time because due to the overwhelming influence of the heuristic reward, the agent does not effectively explore strategies that lead it to solve the problem. Instead, it converges to sub-optimal policies that maximize the heuristic reward but are ineffective or inefficient in terms of the original task.

The obtained results are shown in figure 3. What can be seen, compared to the previous DQN implementation with sparse rewards, is that the total reward function reached a stability level quite early, around 350 episodes. This leads to a rapid increase in the success rate per episode, with small amount of failures, due to the exploration ϵ policy. The success is achieved before 1000 episodes iterations. Additionally, the reward decreases over the episodes stabilizing and low levels after 1000 iterations. This means that it manages to minimize the difference between the predicted Q-values (estimated future rewards) and the target Q-values. The loss behaves as expected because it starts quite high, with a spike when the target network is updated, and then stabilizes at lower values when the reward stabilizes too.

D. DQN with non domain-specific reward

Heuristic rewards are effective but not generalizable. Therefore, to encourage exploration in continuous environments, we implement a Random Network Distillation (RND) algorithm. In this context, a fixed target network and an adaptable predictor network calculate rewards based on the new states, driving the agent to discover new areas [2]. The description of the algorithm can be found in the appendix.

Additionally, the implementation involves normalizing the squared difference between the predictor and the target. This is done by using a running estimate of their mean and standard deviation. The state `s'` is normalized in a similar way. This normalization is crucial because it helps maintain a consistent reward scale across various environments and times, which simplifies hyperparameter selection. This is particularly important given that the target network's parameters are fixed and cannot adapt to varying scales.

Another key setting for this algorithm is the reward balancing parameter, i.e. `reward_factor`. This value is multiplied to the intrinsic reward, which is then added to the original reward to determine the total reward. The `reward_factor` must be adjusted to either increase or decrease its influence on the total reward. A higher `reward_factor` makes the agent behave more like a random agent, whereas a lower one makes it behave more like a standard `DQNAgent`. The appropriate range for the reward balancing parameter is between 100 and 1000. This range enhances the reward's magnitude, making it comparable to the original and impactful, ensuring intrinsic rewards fall between 0 and 100.

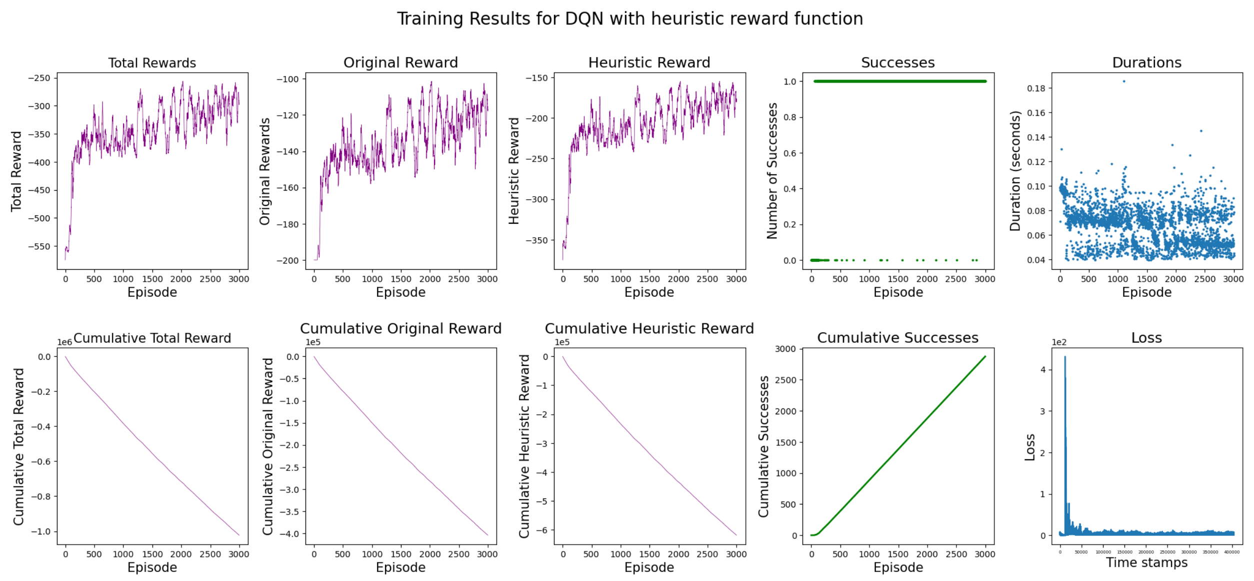

Fig. 3: DQN with heuristic reward results

A balance has been achieved with a `reward_factor` of 700, the effects of which are depicted in Figure 4. The data reveals that the DQN agent with the RND intrinsic reward system initially takes a little longer, compared to the heuristic reward case, to start achieving higher rewards. Additionally the reward presents higher variability. This variability comes from from the random behavior the agent is encouraged to explore, resulting in diverse performance across episodes. As the agent explores more of the environment, the intrinsic reward gradually decreases, reflecting a lower number of new states.

Despite a relatively high number of successes, the algorithm also shows a notable increase in the number of episodes where the goal is not achieved. This outcome suggests that while RND enhances exploration by encouraging the agent to visit new states, it may also interfere with consistent goal achievement by prioritizing exploration over exploitation.

The loss graph shows two spikes: an initial spike followed by a drop, reflecting adjustments to network weights, and a second similar pattern as the agent adapts to new states. After 1500 episodes the loss reaches a stabilization point, similarly to the reward behaviour.

In solving the problem the Prediction-Based Reward approach can be considered. This method is straightforward but can lead to misaligned incentives as rewards are based on prediction errors rather than direct task objectives [3]. Potentially resulting in sub-optimal policies if the prediction model is inaccurate. However, it excels in scenarios requiring rapid exploration with minimal computational overhead.

On the other hand, the DQN with RND approach provides a more balanced strategy between exploration and exploitation, significantly enhancing the learning stability through the use of fixed target networks. This method, though robust, involves a complex, resource-intensive setup that can slow down training processes. The selection between these methods should consider factors such as computational resources, the complexity of environment dynamics, and the necessity for quick versus deep exploration.

IV. Dyna Agent

We will employ a model-based reinforcement learning strategy to solve the given problem. Specifically, we utilize the Dyna architecture, which builds a model of the environment from observations to formulate an effective policy. It is essential to recognize that Dyna is tailored for discrete tabular state spaces. Consequently, our initial task will be to discretize the state space to facilitate the application of the Dyna algorithm.

A. *Overview of the method*

(see pseudo-code in Appendix A)

The Dyna-Q method enhances Q-learning by integrating model-based components to expedite learning and decision-making. (The overview of the method can be found in Appendix C or in the original paper [4])

This model-based method is advantageous for its ability to efficiently explore the state-action space beyond direct experience, leveraging simulated experiences to reduce the dependency on real data acquisition.

B. *Results*

(cf. figure 5) The hyperparameters are:

- $disc\_step$: size of the bins for discretizing the states (0.025,0.05) for (position, velocity)
- $\gamma = 0.99$: discount factor
- ϵ: for the ϵ-greedy policy. Implemented with a schedule initialized at 0.9 and exponentially decaying to 0.05.
- k: number of updates to perform during the planning step
  The final testing performance is 80.2% of success over 1000 new episodes (after 3000 training episodes)

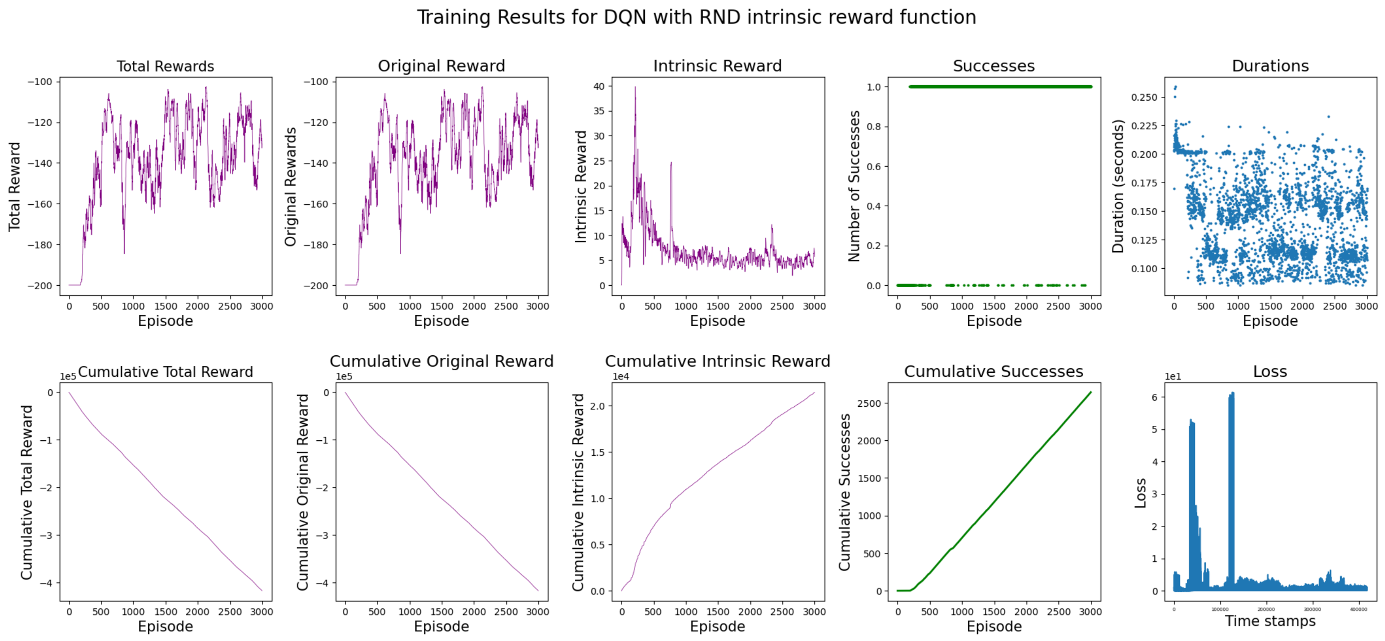

Fig. 4: Results for DQN with non domain-specific reward

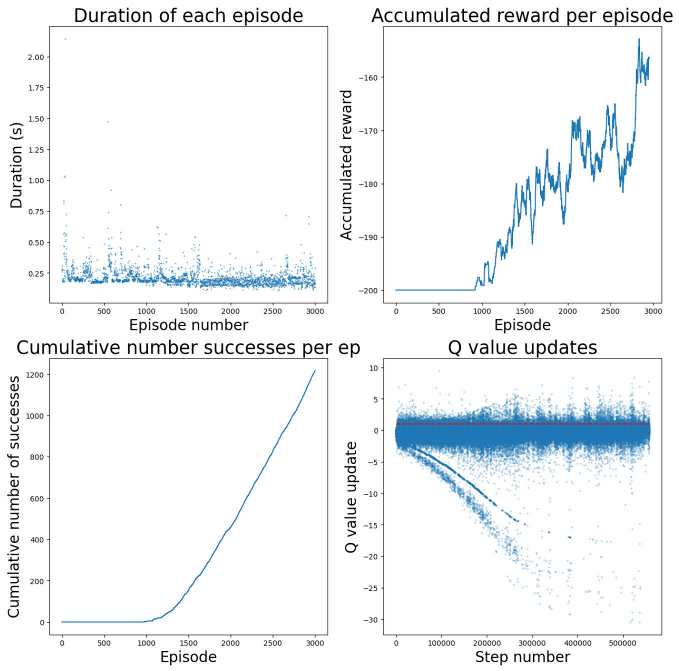

Fig. 5: Dyna training results

1) Influence of the discretization steps: See results in appendix in Figure: 8 and Figure: 9

Obviously, the more the discretization step increases, the longer is the episode, $O\left(\frac{1}{d_1 \cdot d_2}\right)$ where d1 and d2 are the two discretization steps. However, if we use overly sized bins, the agent does not carry out enough details of the environment and so the training is less efficient. Indeed, the intricacies of the slope, position and velocity won't be grasped by the agent. Over 1000 new episodes after 3000 episodes of training, the testing performance is 100% of success for the more detailed environment (steps size divided by 2), and only 34.3% for the less detailed one (steps size multiplied by 2). However the training duration is 3 times the one for the default steps size). From now on, we'll keep up with the default discretization parameters (though it's not the best performing one): the steps are 0.025 for the position, 0.05 for the velocity.

2) Training analysis:

- Q_{values} **update step**, Figure 5 : The Q_{values} updates seem to be coherent throughout the training process. They are mostly negative; which is relevant since the reward is rare and so the agent mostly sees no reward and so the Q_{values} descrease. Also, some steps yield stronger Q_{values} updates. We distinguish a special type of those updates; the low negative ones. It can be the case when the agent is very close to the goal and takes a wrong action, then it goes far from the goal and the Q_{values} associated with the new state will be much lower.
- **Why the DynaAgent manages to solve the task even without any auxiliary reward ?** As the reward is rare, the agent only receives negative reward (-1) and so deacreases the Q_{values} of $(state, action)$ pairs that have already been encountered. The planning step reinforce this behaviour by deacreasing again those values. Hence it encourages the agent to explore states and actions that have not been seen a lot.

3) Learned Q_{values} and trajectories: : see Figure 11

The spiral contour lines likely reflect the car's back-and-forth movements to gain momentum. As the agent learns, it refines its policy to optimize these oscillations. Early in learning, the movements might be erratic and less efficient (broader contours), but as learning progresses, the movements

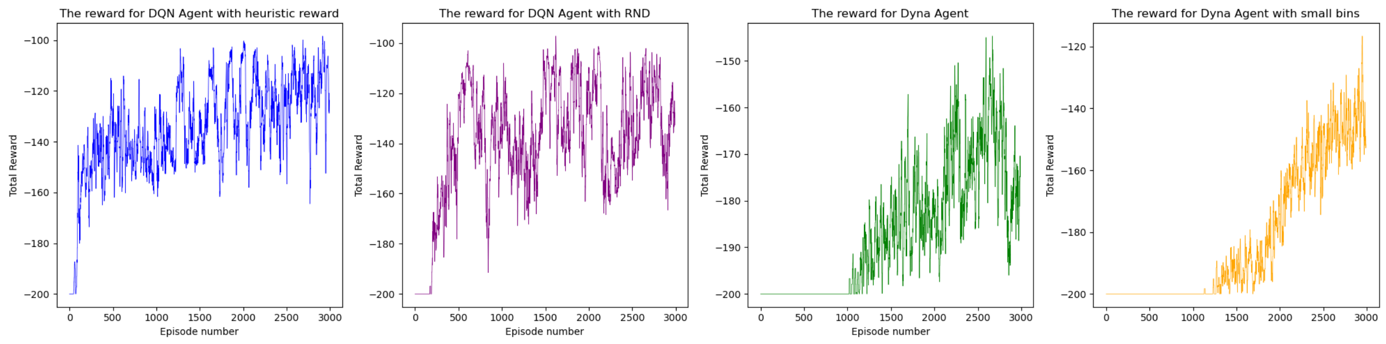

Fig. 6: Comparison of the training reward of the 4 different agents.

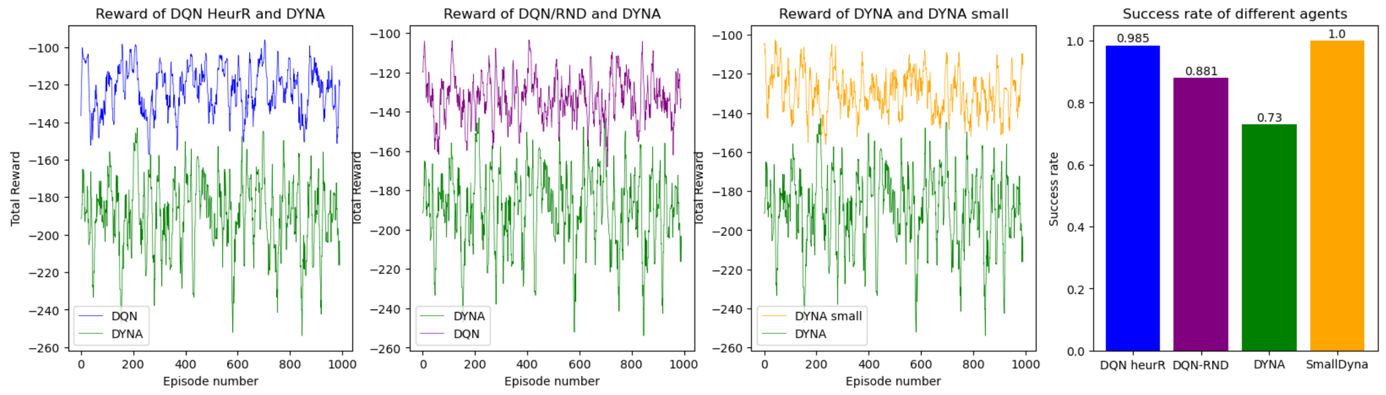

Fig. 7: Comparison of the testing performance of DYNA and DYNA with small sized bins and the two versions of DQN (with auxiliary reward and with non domain-specific reward.

become more focused and efficient, leading to tighter contours around the goal area. Moreover, higher velocities farther from the goal (either left or right of the valley) still have relatively high Q_{values}. This is because high velocity is crucial for the car to ascend the hill. The car needs to accelerate down the slope to gain enough kinetic energy to climb the opposite slope, which explains the high Q_{values} at higher velocities. This learning behaviour is shown with Q_{values} maps at different episodes of the training in the in Figure: 10.

V. Comparison

Training performance: Results are shown in Figure 6. The DQN Agent never achieves success during its training, helping the agent with an auxiliary reward makes a huge difference. From the 350th episode it starts to succeed. The behaviour is very similar for the DQN Agent with RND, though the cumulative reward per episode is less stable. This suggests that the RND technique is similarly effective in enhancing the learning capabilities of the DQN Agent in this environment. When it comes to Dyna Agent, the training behavior totally changes. Indeed, the agent fails until about 1000 episodes. Then it starts seeing a positive reward and the success rate gradually increase. The Dyna Agent, while also variable, shows a somewhat more consistent improvement trend. However we see that in average, the DQN agents performs better (around -130), they gain a higher reward than the Dyna one (around -170 at the end).

Testing Performance: Results are shown in figure 7. It is observed that when tested in the same environment, the Dyna agent consistently underperforms compared to both the DQN heuristic agent and the DQN with RND agent. The DQN with heuristic function not only has a higher average reward but also achieves a significantly greater success rate of 0.985. This superior performance can be attributed to the enhanced decision-making capabilities provided by the ad hoc reward function. However, reducing the bin size in the Dyna algorithm markedly improves its performance, even surpassing the rewards obtained with the heuristic approach and achieving a success rate of 100%. Nonetheless, this adjustment increases the training time.

VI. Conclusion

The project compares the behavior of different types of Deep Q-Network (DQN) based agents (with heuristic and non-domain specific rewards) and the Dyna-based agent (with different discretization bin size). The findings reveal that the Dyna algorithm outperforms the DQN with heuristic rewards only in scenarios where the bins are finely discretized, despite the increased training time and costs associated with this setting. Conversely, when the bin size is larger, the DQN with heuristic rewards proves to be the most effective agent, exhibiting slightly better performance than the DQN with RND agent. However, it should be noted that the DQN with RND demonstrates greater generalizability across different scenarios.

REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[2] Or Rivlin. Reinforcement learning with exploration by random network distillation, 2019.

[3] Sarah Bechtle, Bilal Hammoud, Akshara Rai, Franziska Meier, and Ludovic Righetti. Leveraging forward model prediction error for learning control, 2020.

[4] Richard S. Sutton, Csaba Szepesvari, Alborz Geramifard, and Michael P. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping, 2012.

[5] Zhe Zhang, Yukun Zou, Junjie Lai, and Qing Xu. M^{2}dqn: A robust method for accelerating deep q-learning network, 2022.

Appendix

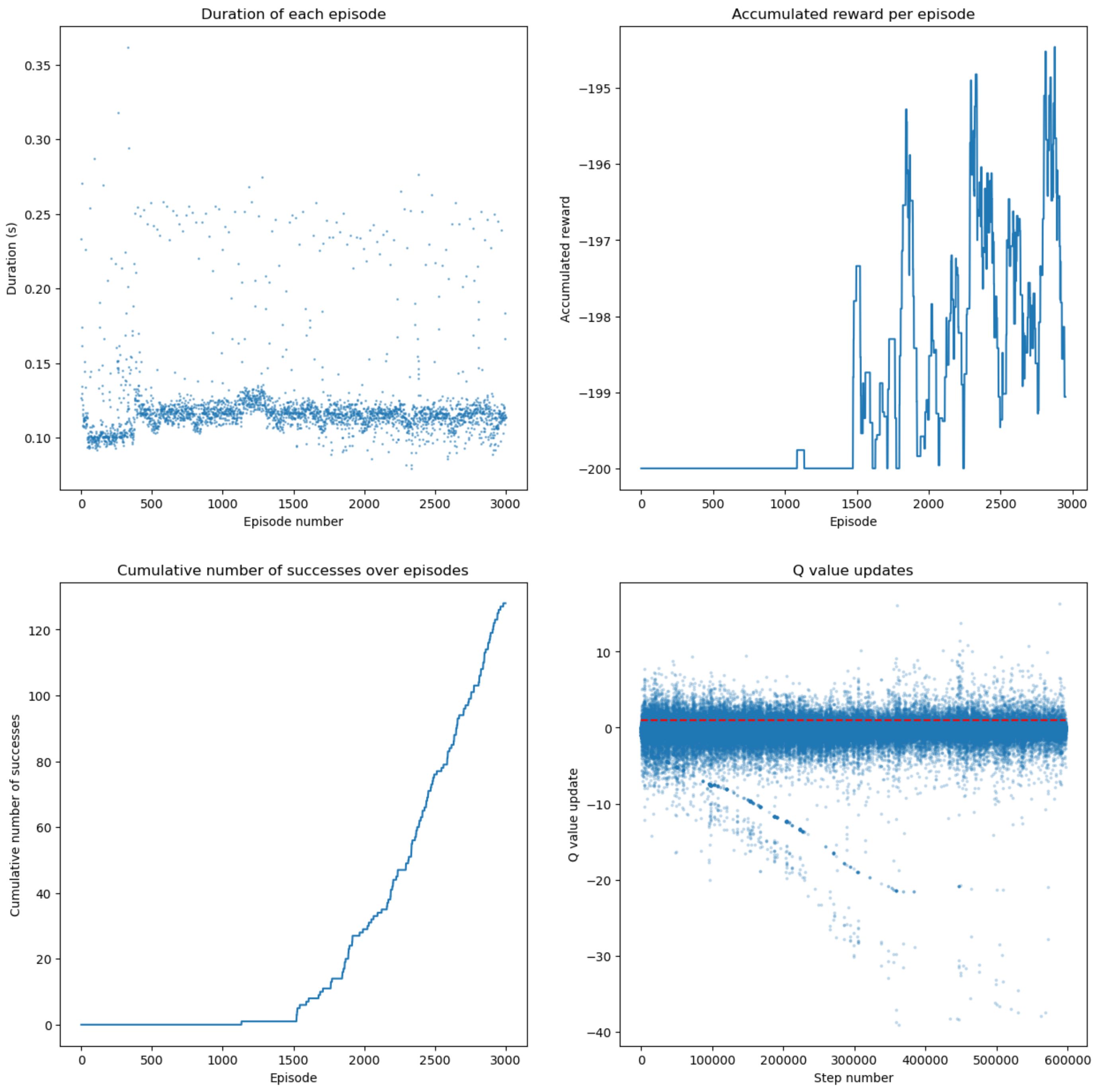

Fig. 8: Results Dyna agent with big discretization steps (0.05,0.01)

A. *DQN: Overview of the method:*

In order to approximate the optimal action-value function $Q(s, a)$, the algorithm uses a deep neural network which estimates the expected future reward for taking action a in state s and following the optimal policy thereafter. The neural network takes the current state s as input and outputs Q-values for all possible actions.

As described in [5], to stabilize the training process, DQN uses two separate networks: the Q-network and the target network. The target's weights are updated less frequently than the Q-network, providing a more stable target for the Q-network to converge to. During training, the agent's experiences (s, a, r, s') consisting of the state, action, reward, and next state are stored in a replay

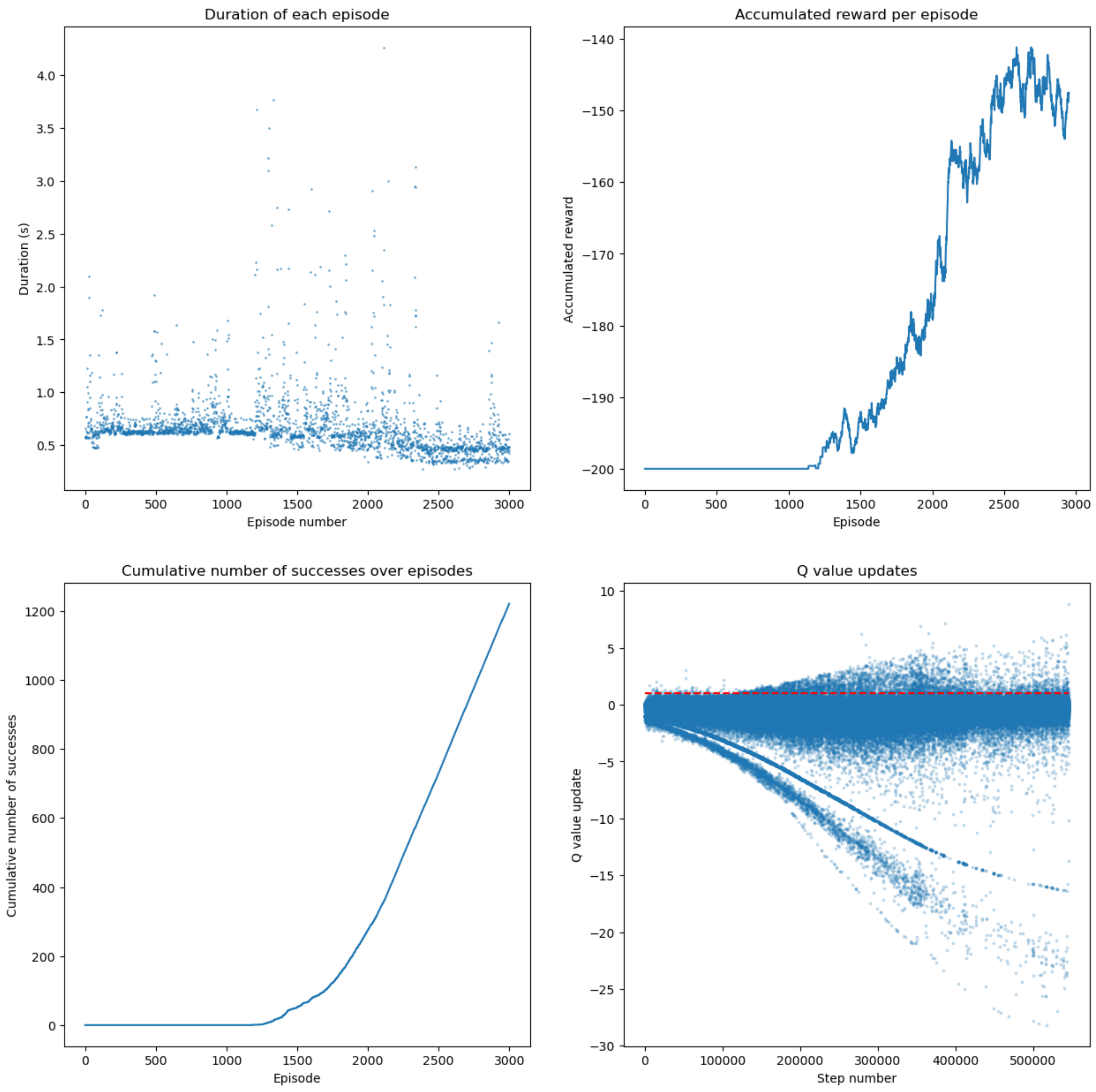

Fig. 9: Results Dyna agent with small discretization steps (0.0125,0.0025)

memory. Random batches of experiences are sampled from this memory to update the Q-network, breaking the correlations in the sequence of observations and improving data efficiency.

The loss function for the Q-network is based on the temporal difference error between the predicted Q-values and the target Q-values computed using the Bellman equation and the target network. The Q-network is trained by minimizing this loss function using an optimization algorithm like AdamW. Finally, DQN uses an epsilon-greedy policy for exploration, where the agent chooses a random action with probability ϵ, and the action with the highest predicted Q-value with probability $1 - \epsilon$. The value of epsilon starts from $\epsilon = 0.9$ and is decayed over time by $decay = 0.995$ until in reaches a minimum of $\epsilon = 0.1$. The decay allows to shift from exploration to exploitation over the episodes.

Algorithm 1 Tabular Dyna-Q

```
1: Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in S$ and $a \in A(s)$
2: loop
3:     $S \leftarrow$ current (non-terminal) state
4:     $A \leftarrow \epsilon$-greedy$(S,Q)$                                      ▷ Q-learning
5:     Take action $A$; observe resultant reward, $R$, and state, $S'$
6:     $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
7:     $Model(S,A) \leftarrow R, S'$                                          ▷ Model Update
8:     loop repeated $k$ times                                                ▷ Planning Step
9:         $S \leftarrow$ random previously observed state
10:        $A \leftarrow$ random action previously taken in $S$
11:        $R, S' \leftarrow Model(S,A)$
12:        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
13:    end loop
14: end loop
```

B. *DQN with non-domain specific rewards: Overview of the method*

In order to implement this algorithm, two additional networks, the predictor and the target network of the RND, are integrated into the existing DQN framework. If the DQN's target network updates itself using Bellman equations to estimate the best future rewards, the RND target network stays the same with its random weights fixed. This design ensures that it consistently assigns high rewards for actions in unfamiliar states, incentivizing the agent to explore parts of the environment that are less visited or unknown. By doing so, it enhances the overall learning and exploration strategy of the agent.

C. *DYNA: Overview of the method*

- **Model Construction:** Dyna-Q constructs models for the transition probabilities T and the expected rewards R. Here, $T(s,a,s')$ denotes the probability of transitioning from state s to state s' after taking action a, and $R(s,a)$ denotes the expected reward for taking action a in state s.
- **Simulating Experiences:** This involves generating synthetic experience tuples to enhance learning efficiency by using fabricated scenarios for Q-table updates, thus maximizing learning from each real interaction.
- **Q-Table Update:** The Q-table is updated using both real and hallucinated experiences, similar to updates in standard Q-learning.

To refine T and R, Dyna-Q utilizes observed transitions and rewards:

- **Learning T:** A frequency table T_c records the occurrences of state transitions for each action. Transition probabilities are estimated by:
$$T(s,a,s') = \frac{T_c(s,a,s')}{\sum_i T_c(s,a,i)}$$
where i runs over all possible resulting states from state s and action a.
- **Learning R:** The reward model R is updated incrementally, akin to updating the Q-table, by averaging observed rewards.

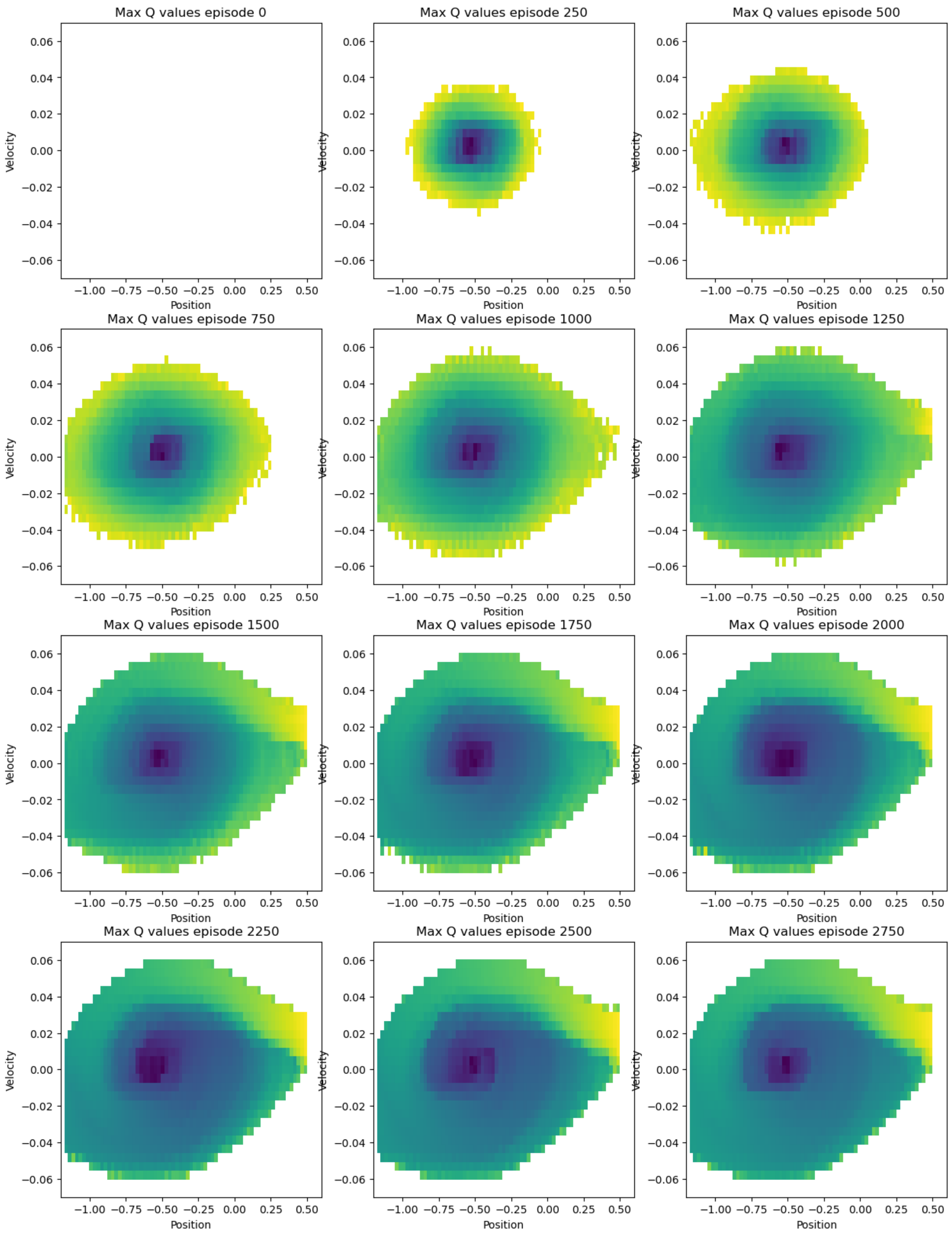

Fig. 10: Evolution of max Q_{values} during training

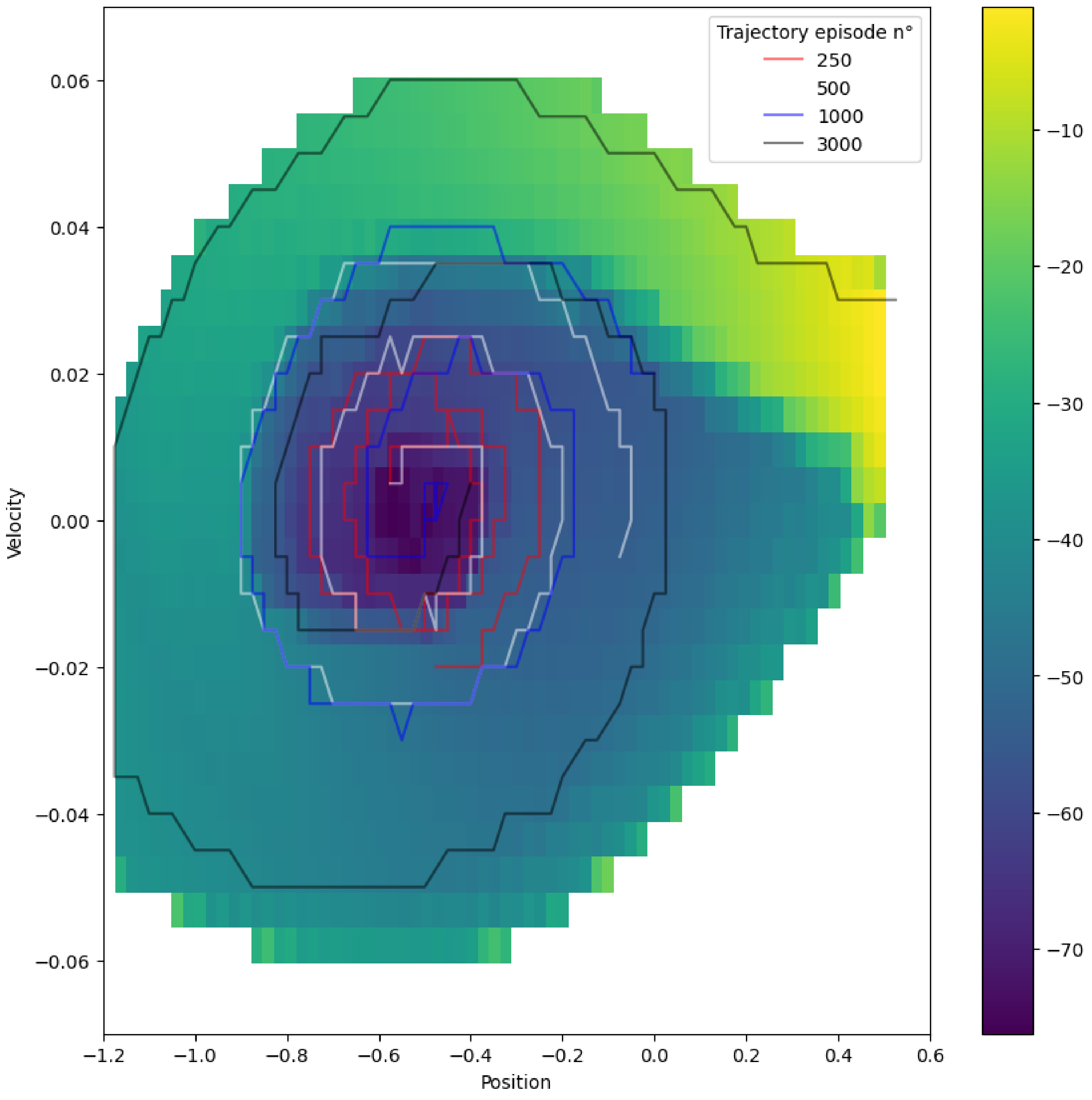

Fig. 11: max Q_{values} after training and episode trajectories