Mini-project 1: Dealing with sparse rewards in the Mountain Car environment

Marc Schenk [1] **Romino Steiner** [1]

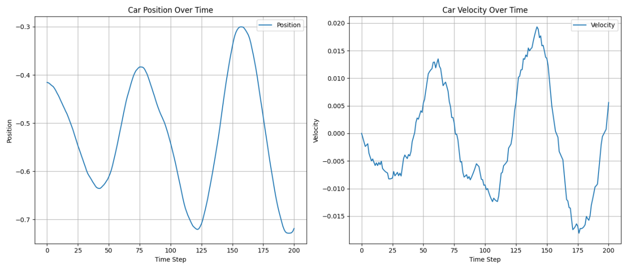

Figure 1: Random Agent on a single episode

1. Introduction

This project considers the Mountain Car environment, detailed in the project description. Such as to find a solution to its problem statement, different reinforcement learning approaches are implemented and compared based on their performance in the Gymnasium environment, as documented in the link provided in the project description.

Prior to introducing our first reinforcement learning agent trying to solve the problem, we note that each of our agents is implemented as a Python class and possesses the following 3 functions at least: (1) `observe(self, state, action, next_state, reward)`: called whenever a new transition of the environment is observed. (2) `select_action(self, state)`: picks the next action from a given state. (3) `update(self)`: performs training after each environment step.

2. Random Agent

To gain initial insights, we run a random agent that makes decisions randomly without learning. Its implementation, as it will be the case for every other agent to follow, can be found in the Jupyter notebook accompanying this project report.

Figure 1 shows the car's states for a single episode with at most 200 steps. The car moves aimlessly, making no progress as expected since no learning occurs.

We extended the experiment to 100 episodes, examining the duration of each. Recalling the Gymnasium documentation, we realize that an episode ends either when the goal state is reached or when the episode length reaches 200. Figure 2

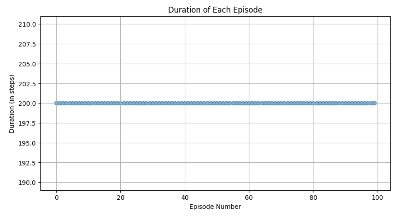

Figure 2: Random Agent episode duration for 100 episodes

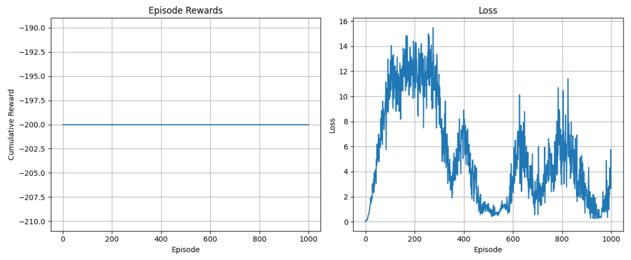

Figure 3: DQN Agent episode rewards and loss for 1000 episodes

shows that all episodes were truncated at 200 steps, indicating the difficulty of reaching the goal state with a random policy.

3. DQN

Having failed to conquer the mountain car environment using the random agent, we shall now explore more sophisticated agents. To that end, we implement a DQN agent.

3.1. Implementation

At a high level, DQN combines Q-learning with deep neural networks to approximate the Q-value function for high-dimensional state spaces, using experience replay and a target network to stabilize training. Thus, the DQN agent features two MLPs and a replay buffer for sampling previous experiences. These attributes enable the DQN agent to handle the continuous state space of the environment. To balance exploration and exploitation, we use an ϵ-greedy policy with an exponentially decaying ϵ. For more implementation details and parameter values, we kindly refer our readers to the Jupyter notebook accompanying this report.

3.2. No auxiliary reward

Training the basic implementation of DQN to solve the mountain car problem proofs to be unfruitful.

As can be seen in Figure 3, equivalently to the random agent,

[1]Master in Cyber Security, ETHZ & EPFL.

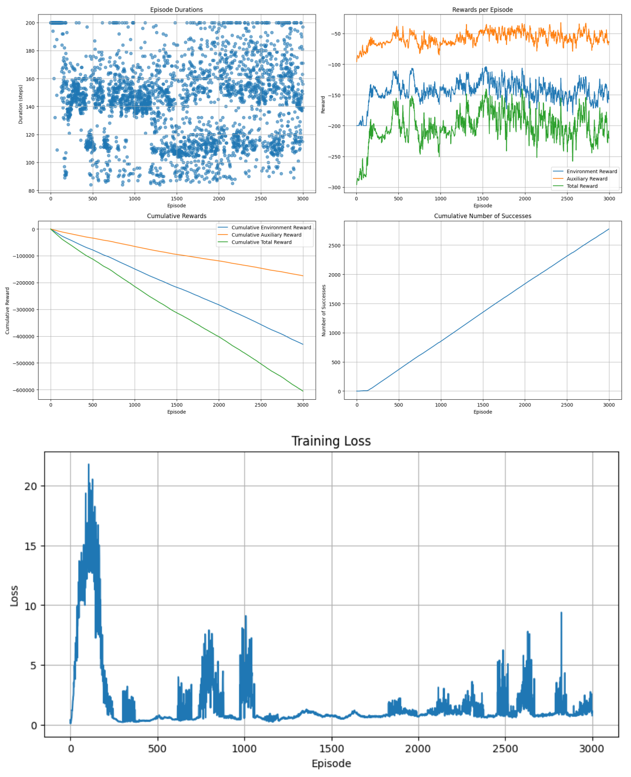

Figure 4: DQN with heuristic reward evaluation metrics for 3000 episodes, reward scaling = 1.

every episode duration is 200, indicating that even after 1000 episodes the agent was unable to reach the goal. In addition, the loss fluctuates, indicating that no function could be found to reliably estimate the Q-values. We accredit this failure mainly to the sparse reward function present in this environment.

3.3. Heuristic reward function

To tackle the aforementioned problem of the sparse reward function, we extend the DQN agent with an auxiliary reward function. The key idea is simple: To handle the sparse rewards, we calculate a suitable auxiliary reward which is added to the environment reward to obtain a total reward that is then provided to the agent. As an auxiliary reward, we use the agent's scaled position value. Intuitively this makes sense, as the position of the car varies between -1.2 at the point furthest away from the goal and 0.6 at the point where the goal is. In this way, the auxiliary reward encourages the agent to move towards the goal. As can be gathered from Figure 4, this solution reliably solves the task at hand.

Indicated by the straight line for the cumulative successes, the agent consistently solves the task after ~ 150 episodes. Through training the agent with auxiliary rewards of different magnitudes, we found that directly using the position (i.e., a reward scaling factor of 1) yields the best results. For excessively large auxiliary rewards, the agent achieved fewer cumulative successes, while for very small auxiliary rewards, it took longer to consistently reach the target. This is likely due to the auxiliary reward being either negligible compared to the environment reward or overwhelming it. Consequently, an auxiliary reward approximately matching the order of magnitude of the environment reward worked best. In Figure 4, it can be observed that the loss initially increases, then quickly decreases and fluctuates thereafter. This behavior is expected as the agent continues to explore, albeit at a reduced rate.

3.4. Non domain-specific reward

While successfully solving the task, the DQN agent with a heuristic reward function relies on specific environment knowledge, limiting its generalizability. To address this, we use Random Network Distillation (RND) to construct an intrinsic auxiliary reward.

RND uses two neural networks initialized randomly: a constant target network and an adaptable predictor network. During training, the predictor network learns to match the target network's outputs for visited states. The difference in their outputs indicates whether a state has already been visited, encouraging exploration of new states. While registering all visited states is feasible in discrete settings, it is not in continuous settings, making RND an effective solution.

All additional functionality needed for RND is implemented as described in the project handout and can be investigated in the Jupyter notebook. One might note that we normalize both the state s' before evaluating the predictor and target network, as well as the squared difference between the obtained outputs. Normalizing the state s' before feeding it into the networks ensures that the inputs to the networks are on a consistent scale, which improves the learning stability of the predictor network. Normalizing the squared difference between the predictor and target network outputs ensures that the RND reward has a consistent scale, preventing it from overwhelming or being overwhelmed by the environment rewards. Neglecting these normalizations could lead to unstable training, where the intrinsic reward signal either dominates or becomes negligible compared to the environment rewards, causing ineffective exploration behavior.

We used a `reward_factor` parameter to balance the environment and RND rewards. Testing revealed that a value of 10 works well. Lower reward factors led to quicker initial successes but less consistent performance and fewer cumulative successes due to insufficient exploration motivation. Higher reward factors caused unstable learning, likely due to excessive focus on exploration. Again, the detailed tests and corresponding plots can be found in the Jupyter notebook.

Comparing the performance metrics of DQN with RND seen in Figure 5 to the ones for the heuristic reward we notice: (1) For RND, the training loss is initially higher due to the predictor network learning, but it quickly stabilizes as

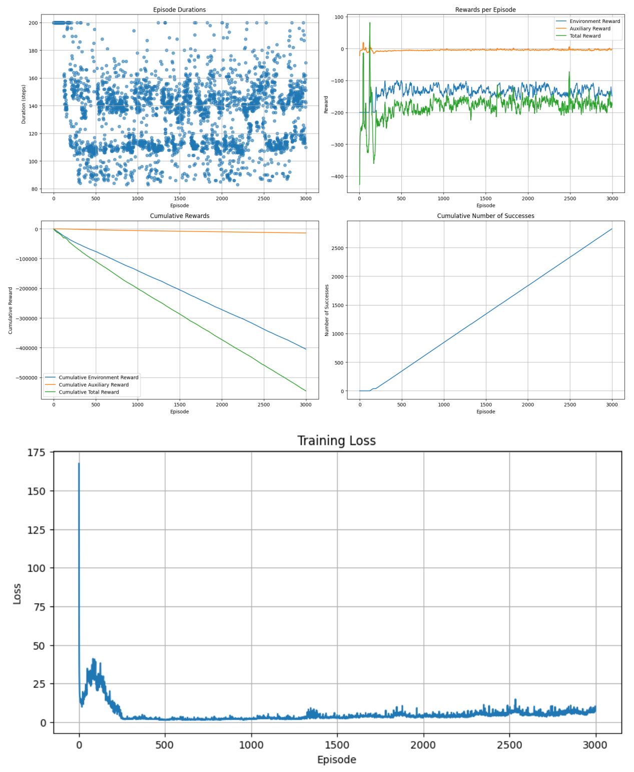

Figure 5: DQN with RND reward evaluation metrics for 3000 episodes, $reward\_factor = 1$.

the predictor becomes more accurate. (2) While with RND the agent initially takes longer to reach the goal, once it does, it reaches it very consistently. Both these findings suggest that with RND the agent explores the environment more thoroughly compared to using a heuristic reward, enabling it to find more optimal policies.

4. Dyna

Having explored the application of the model-free DQN algorithm to the mountain car environment in Section 3, we now delve into a model-based approach to see how its performance holds up. More precisely, we consider the Dyna algorithm as a representative for a model-based approach.

4.1. State discretization, Model building and Implementation

Applying Dyna effectively to the mountain car environment poses two major challenges, which our implementation does overcome ultimately: (1) The problem of discretizing the inherently continuous state space of the mountain car environment such that Dyna becomes applicable. (2) The issue of efficiently building a world model by maintaining an estimation of transition probabilities as well as an estimation of the rewards for each state-action pair.

The entire implementation of our Dyna agent can be found in the accompanying Jupyter notebook. In addition to adhering to the general format introduced in Section 1, the implementation makes use of all the attributes outlined in the report description and introduces supplementary parameters and methods to handle Dyna functionality smoothly.

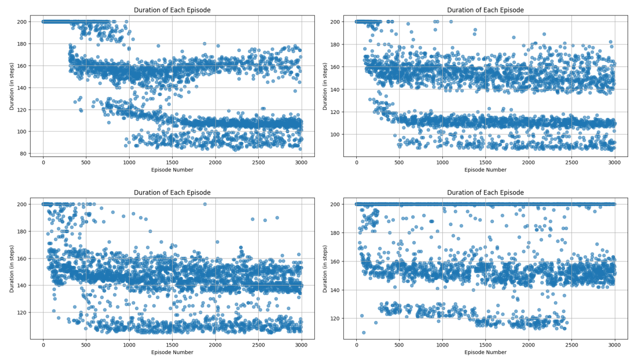

Figure 6: Dyna episode duration with different bin sizes for 3000 episodes, k = 10

4.2. Results

Since we now have an implementation of the Dyna reinforcement learning agent at our disposal, we can start exploring how it interacts with the mountain car environment and whether it is able to learn a suitable policy for solving the problem statement.

We run the agent in the environment for 3000 episodes to observe episode duration trends. As shown in Figure 7 in the episode durations plot, the Dyna agent successfully solves the task over time. According to the Gymnasium documentation, the agent consistently starts solving the task around episode 250. Initially, the agent explores the environment using an $\epsilon -$ greedy policy with a decreasing ϵ schedule, eventually finding valid solutions. These results were obtained with the agent's parameter k set to 10. This value was selected based on an ad-hoc study, detailed in the accompanying Jupyter notebook, which found it balanced runtime and performance well. For brevity, the ad-hoc study is not included in this report, but we will continue using k = 10 for the rest of this section.

Given that the agent can indeed solve the mountain car problem when being exposed to the environment for a sufficient number of episodes, one might want to investigate whether there is some correlation between the bin sizes and the agent's performance. Figure 6 therefore shows the episode duration of training reruns for 3000 episodes with the bin sizes for each of the dimensions increasing steadily. From top left to bottom right, the respective bin sizes are: [0.0125, 0.0025], **[0.025, 0.005]**, [0.05, 0.01] and [0.1, 0.02]. The bin sizes in bold are the ones from the report description. Moreover, the position bin size is given by the first entry and the velocity bin size by the second one.

These experimental results support one's intuition insofar that overly large bins yield a lower number of successes

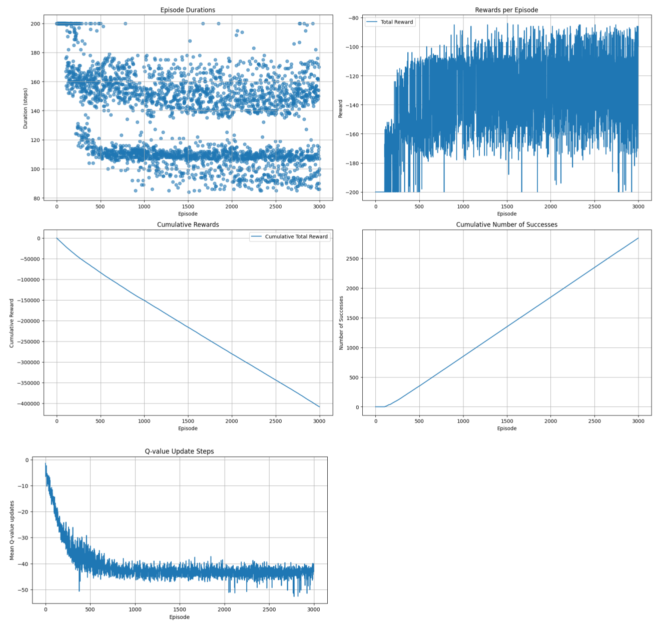

Figure 7: Dyna evaluation metrics for 3000 episodes, `k` = 10

and that too small bins yield more consistent performance after a longer exploration phase. Overly large bins tank performance strongly since they inaccurately abstract the continuous environment space, thereby hindering the agent from trying to precisely locate the goal state. Too small bins on the other hand achieve more condensed performance due to a more fine-grained state space representation, incurring longer exploration but then permitting the choice of more alike actions from the various starting locations. In any case, as the bin sizes are inversely proportional to the size of the occupied memory, small bin sizes are resource-heavy.

Such as to build up an intuition as to how the Dyna agent fares in contrast to the DQN agents introduced in Section 3, we recreate the same plots in Figure 7 for evaluating its performance as we did for DQN. Please note that the training loss is replaced by a plot showing the mean Q-value updates per episode. We reiterate that the episode duration plot clearly shows that after an initial exploration phase of ~ 250 episodes, the agent starts to consistently solve the mountain car problem. This is supported by the rewards per episode plot in light of the absence of auxiliary rewards and the definition of the environment rewards. Moreover, the cumulative rewards and cumulative successes function graphs start to decrease or increase proportionally to the number of episodes respectively after the presumed exploration phase ends, indicating stability and success of the learned policy. Finally, the mean Q-Value update steps start converging after the supposed exploration phase ends, implying that the agent adopts a suitable learned policy thereafter.

Hitherto, it has been mentioned that the Dyna approach is capable of solving the problem at hand without any auxiliary rewards in play, unlike DQN as shown in Section 3.

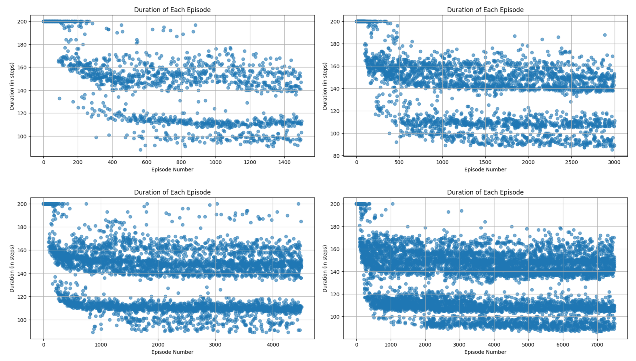

Figure 8: Dyna episode duration for different number of episodes, `k` = 10

This quality of Dyna can be attributed to the fact that it benefits from a positive synergy between direct environment interaction and model-based information from simulated experiences. This speeds up learning in general and overcomes the issue of a sparse reward space when leveraging the construction of a model of the problem domain. Apparently, this is powerful enough to be able to forgo the inclusion of auxiliary rewards.

In most of the figures referred to in the previous paragraphs, one can observe that Dyna displays some sort of pattern with respect to the episode duration over time. To get a more thorough handle on this observation, we rerun Dyna on the mountain car environment for different episode counts and create the respective episode duration plots in Figure 8. The recurring pattern can be described as the Dyna agent mostly solving the problem after the exploration phase of ~ 250 episodes before starting to oscillate between approximately 3 duration ranges until termination. Said behaviour could stem from the car not always being placed in the exact same location in the environment prior to an episode run. Consequently, the agent possibly pursues different action courses based on the bin the starting locations falls into which then converge into one of the observed ranges.

To conclude our experimental analysis of the Dyna agent on the mountain car environment, we analyze the Q-values the agent learns in more depth and relate them to different key episode trajectories showcasing car movement in the space. Drawing on the information conveyed in Figure 9, one can observe that Q-values of positions close to the goal state with a velocity causing the agent to move towards it achieve the highest scores, whereas Q-values of positions in the valley with no movement are attributed the lowest scores. Furthermore, the distribution of Q-value scores takes on a circular structure increasing in a spiralling fashion from the middle to the top right. This is not surprising, as this distribution, when greedily exploited, leads to the car escaping the valley by tilting back and forth appropriately when translated to the mountain car environment.

Additionally, the different trajectories in Figure 9 align well with what can be seen in the episode duration plots, e.g. the

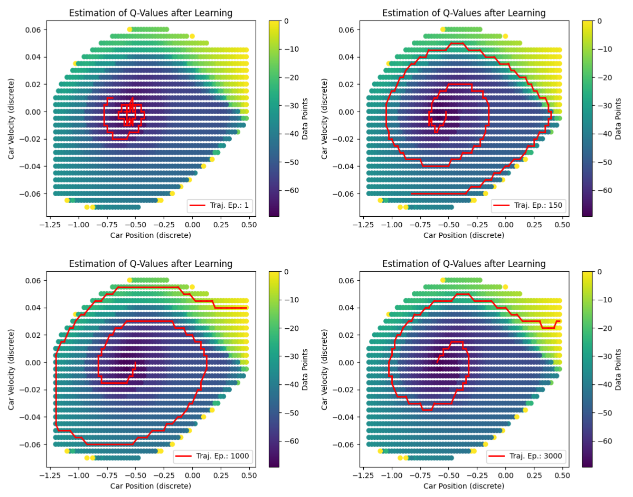

Figure 9: Dyna Q-Value estimation and episode trajectories for 3000 episodes, $\texttt{k} = 10$

one in Figure 7, for the identified key episodes. For the lower episode numbers 1 and 150, where exploration takes place, the agent does not reach the goal and seems to move around somewhat aimlessly or certainly not in an optimal fashion. As the episode number increases, the trajectories show the agent successfully escaping the valley, where some trajectories are shorter and show more optimal runs reflected in lower episode duration values, whilst others cover more discrete spaces, indicating higher episode duration counts.

5. DQN and Dyna Comparison

The purpose of this section is to provide a brief comparison of our three developed agents from Section 3 and Section 4 with respect to the environment rewards they amass over multiple episodes in the mountain car environment.

In Figure 10, we can see how the agents fare in terms of the amassed reward per episode when being trained on the mountain car environment for 3000 episodes (for Dyna, we continue to use $\texttt{k} = 10$ as in Section 4). The following 5 key observations can be made regarding these experimental results: (1) Themes explored in previous sections reappear, such as Dyna only starting to succeed continuously with effective strategies after an initial exploration phase. (2) Dyna offers the most stable performance once exploration ends. (3) Both DQN variants outperform Dyna occasionally in terms of runs achieving low reward counts. (4) A drop in performance occurs after episode 2500 with both DQN variants, where DQN RND seems to recover more easily than DQN with heuristic rewards. (5) DQN RND more consistently achieves lower episode rewards than DQN with heuristic rewards and also amasses less failures.

To create a more comprehensive overview, we also run the pre-trained policies of the three agents for an additional 1000 test episodes, where learning is frozen. From the reward counts per episode shown in Figure 10 for this setting,

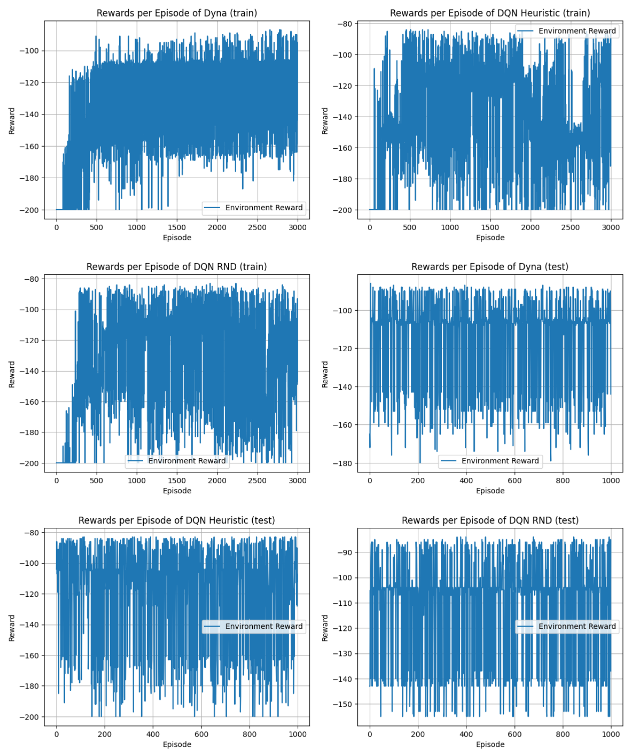

Figure 10: Dyna, DQN Heuristic and DQN RND train and test comparison

3 more interesting insights can be inferred: (1) All agents offer comparable behavior, meaning that environment rewards continuously fluctuate in specific ranges. (2) DQN with heuristic rewards has the best singular episode rewards but is also the only agent to fail at times. (3) DQN RND oscillates in a more compact and overall higher reward range than Dyna which does not quite match our experiences made during the training runs.

6. Conclusion

DQN on its own is not able to solve this task as there are only very sparse extrinsic rewards. Through the addition of heuristic rewards or rewards calculated through RND however, the agent finds the goal quickly and reliably. While heuristic rewards yield faster initial results, the comprehensive exploration facilitated by RND ensures the agent performs more consistently over time.

Dyna on the other hand strongly depends on a seemingly longer exploration phase in contrast to the DQN counterparts for building a model of the mountain car environment. That being said, its method of combining real and simulated experiences provides more stable training performance and overall stable test performance, whilst not being reliant on an auxiliary reward structure. The latter is not possible with DQN in a sparse reward setting as is ours.