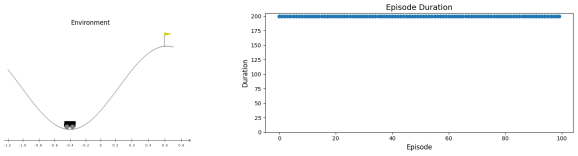
Mountain Car: Handling Sparse Rewards

Jakhongir Saydaliev | 369355 | jakhongir.saydaliev@epfl.ch
Abderrahmane Ould Bay | 330275 | abderrahmane.ouldbay@epfl.ch

1 Introduction

We consider the commonly discussed Mountain Car environment in Reinforcement Learning implemented in Gymnasium [1]. The Mountain Car environment is a deterministic MDP where a car is randomly placed at the bottom of a sinusoidal valley and must strategically accelerate to reach the goal state on top of the right hill. The car's movement is governed by a set of deterministic transition dynamics, with three possible actions: accelerate left, do nothing, or accelerate right (1a). The agent is penalized with a reward of -1 for each timestep, and the episode ends when the car reaches the goal position or after 200 timesteps. The environment's sparse reward function, poses a significant challenge. As seen in 1b, the agent always gets truncated by choosing actions randomly. This report explores the application of model-based DQN (Mnih et al., 2013) and model-free Dyna (Peng et al., 2018) algorithms to overcome this hurdle, as well as extensions of DQN specifically designed to tackle sparse reward problems.



(a) Environment (b) Episode duration of Random Playing

Figure 1: Initial setup of the environment

2 Deep Q-learning

We start by implementing the Deep Q-learning algorithm that is a model-free reinforcement learning algorithm that combines Q-Learning with deep neural networks. It estimates the optimal action-value function, $Q(s, a)$, which represents the expected return when taking action a in state s and following the policy π thereafter. The Q-Learning update rule is modified to use a deep neural network, $Q(s, a; \theta)$, to approximate the action-value function:

$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha \left[r + \gamma \max_{a'} Q_{\hat{\theta}} s', a') - Q_\theta(s,a) \right] \quad (1)$$

where θ and $\hat{\theta}$ are the parameters of the Policy-Network and Target Network, respectively, α is the learning rate, r is the reward received after taking action a in state s, γ is the discount factor, and s' is the next state resulting from action a. [2]

To improve training stability and efficiency, a Replay Buffer D is used to store experience tuples (s,a,r,s'). During training, mini-batches of experiences are randomly sampled from the Replay Buffer to break the correlation between consecutive samples and to reduce variance.

The objective function of the Policy Network is defined as:

$$L(\theta) = \mathbf{E}_{(s,a,r,s')\sim\mathcal{D}} \left[\left(r + \gamma \max_{a'} Q_{\hat{\theta}}(s', a') - Q_\theta(s,a) \right)^2 \right] \quad (2)$$

which we want to minimize by updating the Policy Network parameters, $\hat{\theta}$.

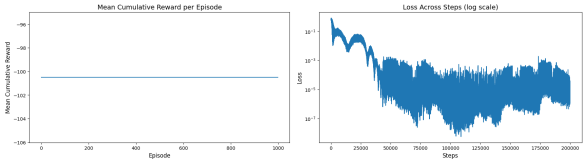
When implementing the update rule of the network we set $\max_{a'} Q_{\hat{\theta}}(s', a') = 0$ if s is the terminal state.

2.1 No auxiliary reward

We implemented the DQN algorithm, as described earlier, and trained the agent over 1000 episodes. Despite the convergence of the loss function, as illustrated in Figure 2b, the agent did not succeed in completing the task 2a. However, a detailed analysis of the reward distribution across the environment, presented in Figure 7a, indicates that the agent is indeed learning, though at a slow pace. This is due to the sparse distribution of reward of the environment, for which we will discuss a solution in the next section by introducing an auxiliary reward.

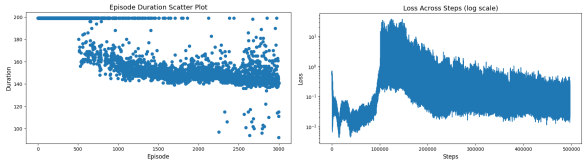
[1] https://gymnasium.farama.org/environments/classic_control/mountain_car/

[2] the hyperparameters we have used is given in A



(a) Mean cumulative reward per episode (b) Training loss (log scale)

Figure 2: Average cumulative reward and the training loss of DQN algorithm



(a) Duration of each Episode (b) Training loss (log scale)

Figure 3: Episode duration and the training loss of DQN algorithm with heuristic reward function

2.2 Heuristic reward function

To help the agent learn faster with intermediate rewards, we add the following reward function to the environment reward that takes into account the current location and the velocity of the agent:

$$r_{aux} = |s'_p - s_{p_0}| + \frac{|s'_v|}{2 * s'_{v_{max}}} \tag{3}$$

Using the reward function specified in 3, we incentivise both high position and high velocity on either side, but we scale the velocity by 0.5 to lessen its impact on the reward. This is because the highest velocity occurs at the bottom of the environment. We also tested various scaling coefficients for both position and velocity, and even considered using only velocity or position. However, in most other scenarios, the agent's learning was significantly slower. Furthermore, a high auxiliary reward can overshadow the actual environment reward, while a very low auxiliary reward might not sufficiently aid the agent's learning.

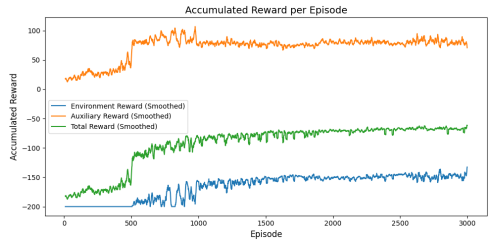
2.2.1 Results

As indicated by 3a, the agent begins its learning process at approximately 600 episodes. According to 4a, in the initial phases, the agent relies solely on the auxiliary reward for learning, which eventually results in the successful completion of the task. Once it completes the task, the agent starts learning from both rewards.

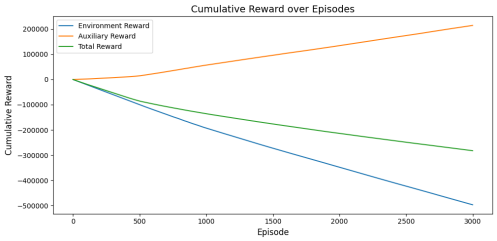
Interestingly, the training loss initially increases when the agent begins to complete the task, and then gradually decreases 3b. This is likely because $Q_{\hat{\theta}}(s', a') = 0$ at the terminal state.

2.3 Non domain-specific reward

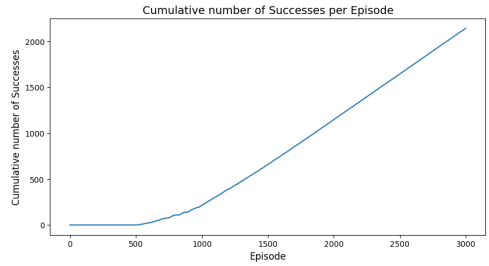
We saw that heuristic reward functions are environment-specific and may not generalize well to different environments. To encourage exploration and overcome this limitation, we explore the usage of Reward Network Distillation (Burda et al., 2018). RND is a technique that addresses the need



(a) Cumulative reward per Episode



(b) Cumulative reward over Episodes



(c) Cumulative number of successes over Episodes

Figure 4: Cumulative Rewards and Success rate of DQN algorithm with heuristic reward function

for an intrinsic reward function in continuous environments. It involves two neural networks: a target network with fixed random weights and a predictor network with adaptable weights. Both networks take a state as input and output a single number. The predictor network is trained to match the target network's outputs on encountered states. The RND intrinsic reward is defined as the squared difference between the predictor and target networks' outputs, normalized accordingly. This reward encourages the agent to explore new areas of the environment, as the predictor network's outputs will be less accurate for rarely encountered states.

Is important to note that before passing to the target and predictor networks, the next state s' is

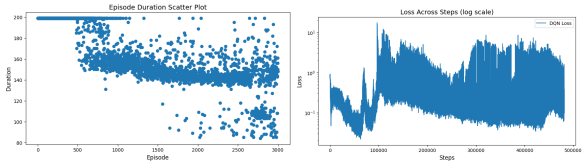
normalized using its running average and standard deviation as $\frac{s'-s'_{mean}}{s'_{std}}$. Because RND is an environment agnostic method and different

Likewise, we compute the RND reward by taking the squared difference between the predictor and target outputs and normalize it using the mean and standard deviation of the RND reward of initial n (n=100) steps and clipped between -5 and 5. We then scale the resulting RND reward by a *reward factor=0.2* [3]. It is all to ensure that the reward is scaled appropriately and does not dominate the environment reward.

2.3.1 results

As indicated by 5a, the agent begins its learning process at approximately 500 episodes which in turn is even earlier than with heuristic reward function 2.2. According to 6a, in the initial phases, the agent relies only on the RND reward for learning, which eventually results in the successful completion of the task. Once it completes the task, unlike to heuristic reward case, the agent starts learning mostly from the environment reward, as we can see the RND reward almost flattens after around 1000 episodes. It is expected as the environment is already *explored* by the predictor network.

Finally, the training loss behaves similar to the heuristic reward case 5b.

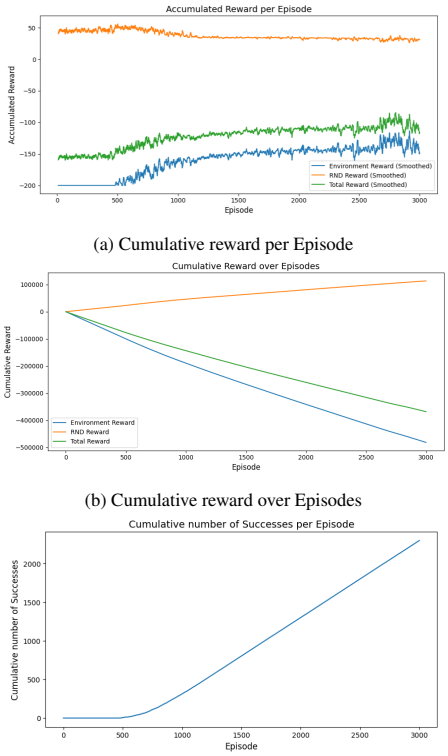


(a) Duration of each Episode (b) Training loss (log scale)

Figure 5: Episode duration and the training loss of DQN algorithm with RND

2.4 Reward comparison of DQN extensions

In 7 we depicted the last 10k rewards from Experience Replay for each Algorithm over the environment states (velocity vs positions). The values of the heat map represent the sum of reward at the corresponding state [4]. It is interesting to see that, unlike RND, with heuristic reward the agent stores the highest reward only at the top of the hill, and the remaining rewards are distributed accordingly to lead up to the hill, which is expected thanks to our definition of the auxiliary reward function 3.



(a) Cumulative reward per Episode

(b) Cumulative reward over Episodes

(c) Cumulative number of successes over Episodes

Figure 6: Cumulative Rewards and Success rate of DQN algorithm with RND

3 Deep Dyna-Q

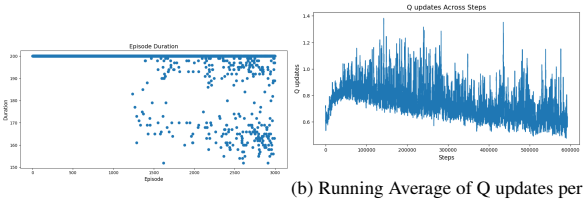
We now present our experiment results using Dyna Algorithm, which is an approach that integrates both model-free and model-based methods by updating the agent's policy using real experience and simulating additional experiences based on a learned model of the environment, enhancing learning efficiency through a combination of direct reinforcement learning and planning; this can be mathematically formulated by iteratively updating the Q-values using both actual observed rewards and simulated rewards derived from the environment model. As our environment is in continuous space, we would first need to convert it into discrete space. We experimented with different bin sizes for position and velocity as:

- *small*: position - 0., velocity - 0.
- *medium*: position - 0.025, velocity - 0.05
- *large*: position - 0.1, velocity - 0.02

[3] We did the normalization always using the initial 100 rewards, instead of a moving normalization, as one of the TAs told us so.

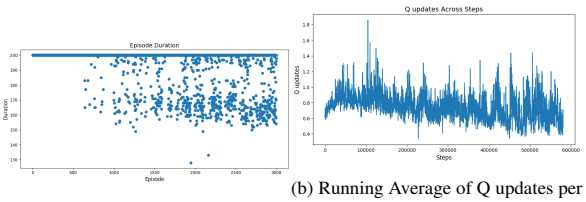
[4] Note that for Basic DQN, the position and the velocity ranges are not given in the full scale.

Choosing larger bins makes the learning process faster as there are less states to learn, but it results in a lower resolution representation of the state space. For example, in our problem, where fine control of momentum is crucial, large bins might prevent the agent from learning the precise actions needed to build momentum effectively 10. While with overly small bins, the learning is slower and computationally costly, but it has more detailed state representation. When we have more states (smaller bins) the interconnection of the states can also be represented better, leading learning the environment better. 8



(a) Duration of each Episode (b) Running Average of Q updates per step

Figure 8: Episode Duration and Q updates using small bins



(a) Duration of each Episode (b) Running Average of Q updates per step

Figure 9: Episode Duration and Q updates using medium bins

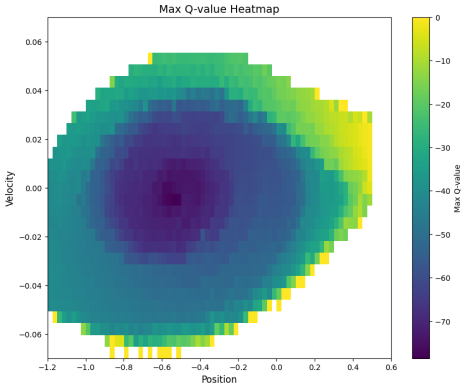
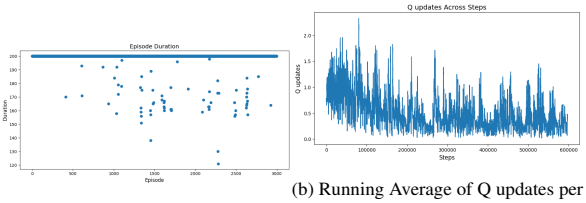


Figure 11: Learned Q-values of Dyna agent

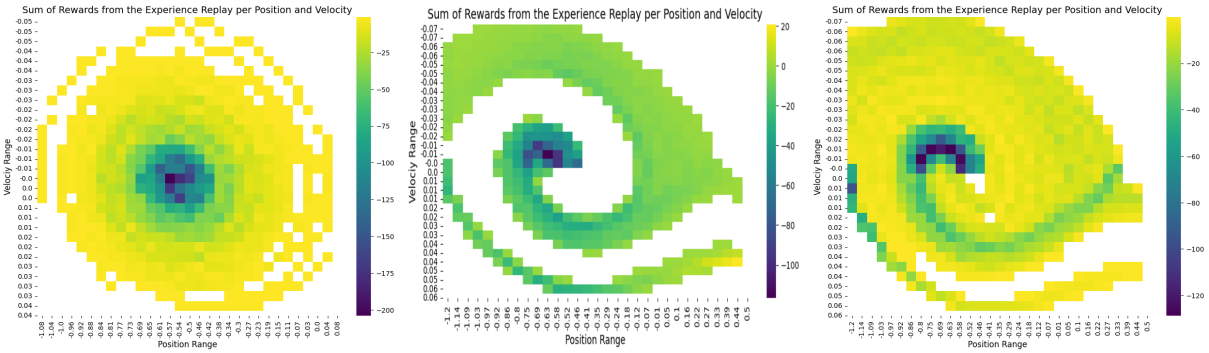


(a) Duration of each Episode (b) Running Average of Q updates per step

Figure 10: Episode Duration and Q updates using large bins

As shown in 9 Dyna algorithm can effectively solve the task, as it makes use of both actual observed reward and a simulated reward. Those simulated rewards help the agent explore the environment faster. Also from 11 we can see that the agent has explored a wide coverage of the state, unlike DQN with Heuristic reward function, and, as expected, it has a high Q-value for the state of the top position (*position*=0.5), and also for high velocities.

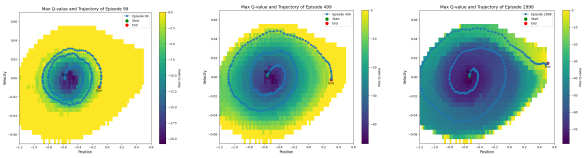
From 9a we can see that the agent starts solving the task frequently after around 1000 steps, and it is probably due to the fact that over time model has



(a) Basic DQN on 1000 episodes (b) Heuristic Reward Functions on 3000 episodes (c) RND on 3000 episodes

Figure 7: Sum of the last 10k rewards from Experience Replay at the corresponding states for 3 algorithms: Basic DQN, Heuristic Reward, RND

already explored a wide coverage of the state as can be seen in 12 as well.



(a) 99 Episodes (b) 499 Episodes (c) 2998 Episodes

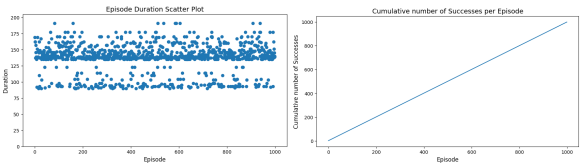
Figure 12: Learned Q values and the trajectory during training at 3 episodes [5]

4 Final Comparison

When we test the above algorithms, we can see that they all have learned the environment and constantly finish the task. Interestingly, all algorithms appear to complete the task in approximately 2 duration points. For instance, the RND agent typically accomplishes the task in about 90 to 150 steps 14a. It means, the agent could have leaned 2 types of policies during training, that could have resulted from the balance between exploration and exploitation stages.

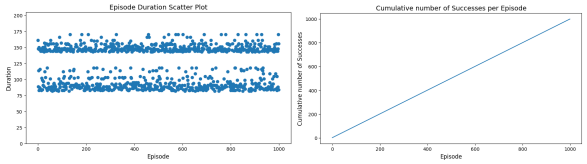
As we saw during in this report, DQN uses a deep neural network to approximate the Q-value function, learning from real experiences by updating the network parameters based on observed transitions from interactions with the environment. This approach allows DQN to handle large and continuous state spaces efficiently by generalizing from the experiences it has seen. But the pure DQN without any extensions, such as RND or auxiliary reward, may fail to learn the enviroment well in case of sparse reward distribution.

In contrast, Dyna combines model-based and model-free reinforcement learning. It builds an internal model of the environment from the real experiences and uses this model to simulate additional experiences, which are then used to update the value function. This allows Dyna to learn more efficiently by augmenting real experiences with simulated ones, effectively increasing the amount of data available for learning.



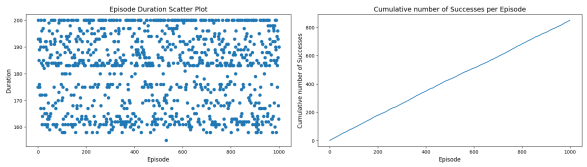
(a) Duration of each Episode (b) Cumulative Reward over Episodes

Figure 13: Playing with Heuristic Reward Function



(a) Duration of each Episode (b) Cumulative Reward over Episodes

Figure 14: Playing with RND



(a) Duration of each Episode (b) Cumulative Reward over Episodes

Figure 15: Playing with Dyna

References

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. 2018. Exploration by random network distillation.

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

Baolin Peng, Xiujun Li, Jianfeng Gao, Jingjing Liu, Kam-Fai Wong, and Shang-Yu Su. 2018. Deep dyna-q: Integrating planning for task-completion dialogue policy learning.

A DQN hyperparameters

We set the following hyperparameters when training the Deep Networks in Deep Q-learning: $\gamma = 0.99$, exponentially decaying exploration with $\epsilon_{start} = 0.9$ and $\epsilon_{end} = 0.05$, replays buffer size is 10000, and we started sampling from the buffer after initial 100 samples. We copy the Policy Network parameters θ into the Target Network parameters $\hat{\theta}$ every 20 steps. The Policy network is trained using AdamW optimizer with 10^{-4} learning rate with batch size of 64.