

Exercise 10

Time to Explore: DQN with RND and ICM

As you have learned from the lecture, for sparse reward problems where the agent has to thoroughly explore the state space to get a glimpse of what his task is (sparse reward setting), exploration techniques like ϵ -greedy are often not enough to "stumble over" the desired goal conditions.

This is already the case for relatively simple problems like the MountainCar¹ environment. In this environment, the agent has to strategically accelerate a car back and forth to escape from a mountain valley and arrive at the top of the mountain at the right (depicted in Figure 1). In the default implementation, the reward function is $R_a(s, s') = -1$ for every step until the episode is finished, i.e. after a maximum of 200 time steps or the agent arrived at the goal. As it turns out, using value-function approximators like DQN, this reward definition is not as sparse as it seems on first impressions (try to explain why this might be the reason. Think about how the value estimates of visited vs. not-visited states change over the course of training.). To convert the MountainCar environment into a genuinely challenging sparse-reward problem, we wrote an environment wrapper inside `env.py` and changed the reward function to:

$$R_a(s, s') = \begin{cases} 200.0, & s' \text{ is a goal-state} \\ 0.0, & \text{otherwise.} \end{cases} \quad (1)$$

Your task is to implement *Exploration by random network distillation* (RND)² and *Curiosity-driven Exploration by Self-supervised Prediction* (ICM)³, two state of the art RL exploration methods. In the following, we give a rough sketch of the algorithms and provide a more detailed, textual pseudo-code inside the code skeletons. We also advise you to quickly look at the respective papers for more information.

In short, for both methods, the goal is to have some metric that encapsulates how often the agent has already visited a certain state. Whereas in discrete state spaces, we could just keep count of state visitations inside a table, for continuous state spaces this is harder to integrate.

RND The idea of RND is to use a randomly initialized, fixed *target* network and a second *predictor* network, where the predictor learns to predict the outputs of the *target* network based on states as input. Even though the *target* network outputs are random, they are deterministic (stay the same) for every state input. This way, the more often the agent has already visited a certain state, the better the *predictor* will be able to approximate the *target* network outputs. In RND, the intrinsic reward is then defined as the error between the prediction and the target outputs.

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

²<https://openreview.net/forum?id=H1lJjnR5Ym>

³<https://pathak22.github.io/noreward-rl/>

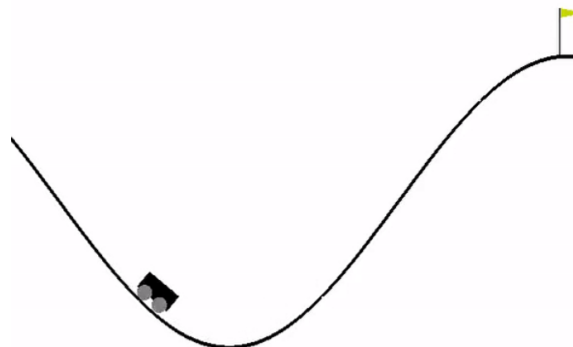


Figure 1: The *MountainCar-v0* OpenAI gym environment.

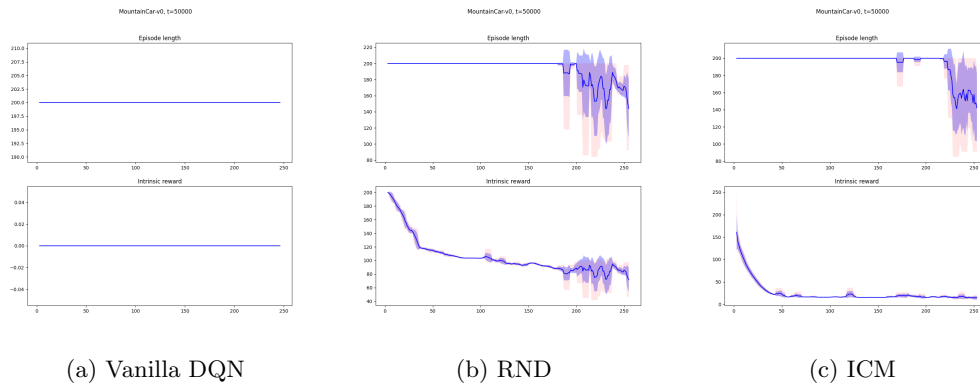


Figure 2: Example return plots for this exercise.

ICM The goal of ICM is to quantify the novelty of states through a forward model. Based on the current observation and the performed action, ICM predicts the next observation, and uses the error between the actual next observation, and the prediction as an intrinsic exploration reward. Because learning a forward model inside the observation space has many drawbacks (think about visual observations), ICM performs the forward prediction inside a learned feature space. To do this, a *feature* network encodes the observations into a low-dimensional feature representation. A second network, the so-called *inverse-dynamics* network, is trained to predict the feature encoding of the next observation based on the feature encoding of the current observation and the performed action. This way, only action-relevant features and stochastic effects outside the agent's control (think about the noisy TV problem) are filtered out. Finally, the *forward-dynamics* network predicts the feature vector of the next observation based on the encoding of the current observation and the performed action. Similar to RND, we use the error as an intrinsic reward.

Programming Tasks

1. Implement RND and ICM inside `exploration.py`. You will have to implement the `calculate_loss(...)` and `calculate_reward` functions of both exploration modules.
2. Run your implementation and verify that they are working. For reference, Figure 2 displays result plots of our solution.

Some Notes

- It can happen that sometimes the methods do not converge. This is due to the fact that these methods are highly dependent on good hyper-parameters. For example, if the RND *predictor* is too powerful and learns too fast, the intrinsic reward becomes non-informative very quickly. This is equally true for ICM. We tried our best to find reliable hyperparameters, but RL can be tricky sometimes. ☺
- If your implementation arrives at the goal sometimes, it is probably correct.
- To highlight the exploration capabilities of RND and ICM, we only use ϵ -greedy for vanilla DQN.