Exercise 12
Time to Explore: DQN, 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 even get a glimpse of what his task is (i.e. the agents only gets a reward when arriving at a goal state), exploration techniques like ϵ-greedy are not enough to "stumble over" the desired goal conditions.

This is already the case for relatively simple problems like the MountainCar[1] 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 located 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.

Task

1. Uncomment the line `env = gym.make('MountainCar-v0')` inside `dqn.py` to run DQN on the MountainCar setup described above. You will see that DQN is able to learn an escape strategy quite quickly. Try to explain why this might be the reason. Think about how the value estimates of visited vs. unvisited states change over the course of training.

As it turns out, using value-function approximators like DQN, this reward definition is not as sparse at it seems on first impression (we will further clarify this in next week's exercise discussion). To convert the MountainCar environment into a truly challenging sparse-reward problem, we wrote an environment wrapper inside `env.py` and changed the reward function to:

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

Task

2. Revert the changes you did and ensure that `MountainCarCustomized` is used as the training environment. Rerun vanilla DQN and verify that it struggles to deal with the modified problem. (Very rarely it might still solve the environment, but very unreliably... ☺)

Now, let's turn to the main part of this week's exercise. 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 exploration methods with not too much complexity. In the following, we give a rough sketch of the algorithms and provide more detailed, textual pseudo-code inside the code skeletons. We also advise you to have a quick 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 (i.e. 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.

[1] `https://www.gymlibrary.ml/environments/classic_control/mountain_car/`
[2] `https://openreview.net/forum?id=H1lJJnR5Ym`
[3] `https://pathak22.github.io/noreward-rl/`

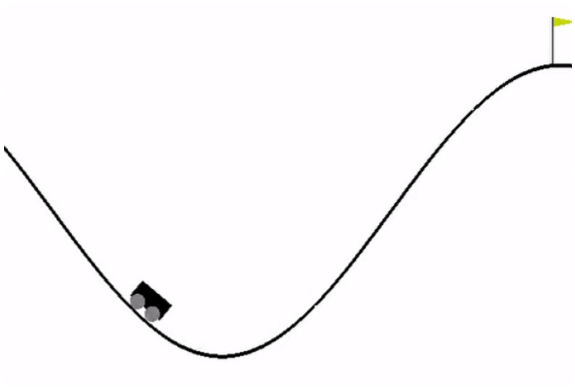
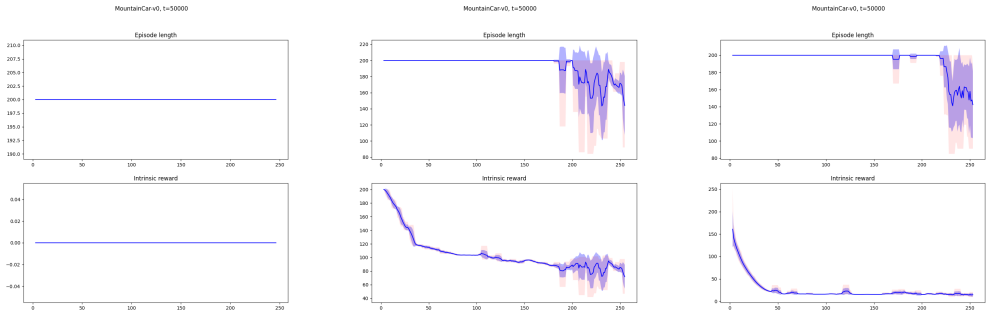


Figure 1: The *MountainCar-v0* OpenAI gym environment.



(a) Vanilla DQN (b) RND (c) ICM

Figure 2: Example result plots for the current exercise.

ICM The goal of ICM is to quantify the novelty of states through a forward model, i.e. 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 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 feature space. 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 are learnt and stochastic effects outside the agent's control (think about noisy-TV) 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 intrinsic reward.

Programming Tasks

3. Implement RND and ICM inside `exploration.py`. You will have to implement the `calculate_loss(...)` and `calculate_reward` functions of both exploration modules.
4. 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 to 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.