Reinforcement Learning Lab

Lesson 11: DRL in Practice

Davide Corsi and Alberto Castellini

University of Verona
email: davide.corsi@univr.it

Academic Year 2022-23

UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Today Assignment

In today's lesson, we will face a classical DRL problem MountainCar. In particular, the file to complete is:

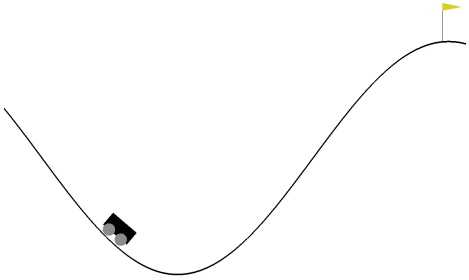
`RL–Lab/lessons/lesson_11_code.py`

Inside the file, you will find empty methods of the A2C algorithm (that you have to fill in with your code) and a class called OverrideReward that you should modify to find the best reward function to solve the problem, in particular, the method *step*. The class and method to complete are:

- **class OverrideReward()**
- **def step(self, action)**

Notice that other changes should be made in order to adapt the algorithms to a different environment (e.g., network shape and hyperparameters); and some additional function implemented (e.g., **createDNN**, **training_loop**, and **A2C**).

Environment: MountainCar



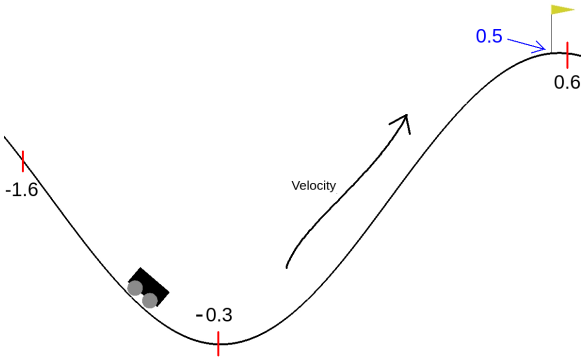
- The Mountain Car problem is a classic reinforcement learning problem with the goal of training an agent to reach the top of a hill by controlling a car's acceleration. The problem is challenging because the car is underpowered, and it cannot reach the goal by simply driving straight up.
- The **state** of the environment is represented as a tuple of 2 values: *Car Position* and *Car Velocity*.
- The **actions** allowed in the environment are 3: *action 0 (accelerate to the left)*, *action 1 (don't accelerate)* and *action 2 (push cart to right)*.

State Space and Original Reward Function

1. The original reward function is *discrete*, specifically a reward penalty of -1 is assigned for each timestep (see the step() function of the original environment here). However, this function does not provide the agent with any intuition on how to reach the goal, resulting in poor performance and a long training time.
2. Our suggestion is to provide a more insightful reward function, to drive the agent toward reaching the goal. To achieve good performance is typically useful to exploit a continuous reward function, that provides insightful information at each step.
3. To design an informative reward function, it is fundamental to understand the observation space and the action space. In our problem, the observation space is of 2 elements:

```
# Get the observation from the environment
observation, reward, terminated, truncated, info = self.env.step( action )
# First observation: POSITION on the x-axis
position = observation[0]
# Second observation: VELOCITY
velocity = observation[1]
```

Understanding the Environment



- The f observation is the position of the car. The values range from -1.6 to 0.6, the central value is 0.3 (see the figure). The task is considered solved if the car reaches position 0.5, i.e., it touches the flag.
- The second observation is the velocity of the car, which assumes values in the range ± 0.07; where positive values mean that the car is pushing on the right and vice-versa.
- More information can be found in the step function of the source code here.

Override the Original Reward

To modify the reward function, in the suggested code we exploit the Gymnasium Wrappers to override the step function. (here the official documentation).

```
# Gymnasium wrapper
class OverrideReward( gymnasium.wrappers.NormalizeReward ):
    # Override the Gymnasium step function
    def step(self, action):
        previous_observation = np.array(self.env.state, dtype=np.float32)
        observation, reward, terminated, truncated, info = self.env.step( action )
        # Here you can manipulate the reward function
        return observation, reward, terminated, truncated, info
```

- The step function of the wrapper constitutes a mask for the actual step function of the environment. Here you must return the same values as the actual Gym step function. The wrapper, however, offers the possibility to intercept the returned values and modify them.
- Insire the step function it is possible to call all the standard gym functions from the python object *self.env*.

Code Snippets and References

Remember to always adapt the network structure to the state and action spaces:

```
observations = env.observation_space.shape[0]
actions = env.action_space.n
```

To better understand the environment a graphical visualization can be useful, following a code snippet (*remember to install pygame via pip*).

```
# Add the flag 'render_mode' to visualize the enviornment
env = gymnasium.make( "MountainCar-v0", render_mode="human")
```

References:

- Additional environment details: here.
- Gym wrappers: here.
- Code example for rendering: here.

Expected Results

Why Episode Length?

Since we are changing the reward function, the direct comparison of the average performance is not interesting. For our problem, a good proxy for the performance is the episode length (in some other cases, a typical valid option is the success rate).

Performance

Without any change in the reward function, the agent is unlikely to solve the task. If your agent reaches the goal (i.e., **less than 1000 steps**) the task can be considered solved.

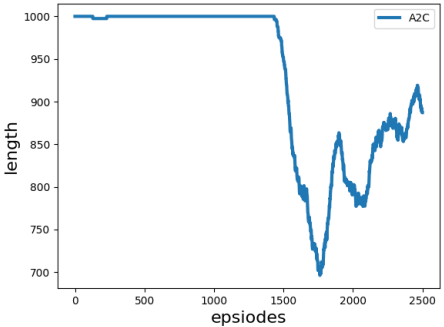
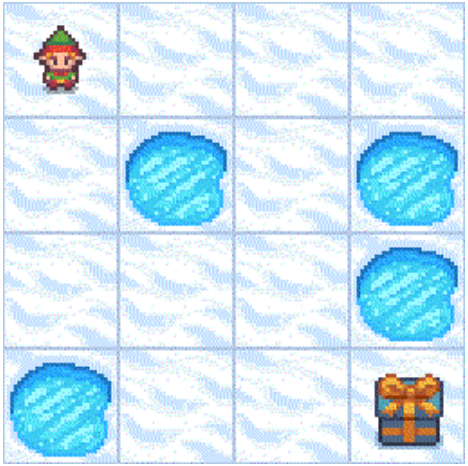


Figure: Note that obtaining this result requires time. You can stop the training as soon as the task is solved (i.e., first length below 1000 steps).

Additional Exercise: Frozen Lake (pt. 1)

- Frozen lake involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. The player may not always move in the intended direction due to the slippery nature of the frozen lake.
- The **state** of the environment is represented as a single integer representing the player's current position as current_row * nrows + current_col (where both the row and col start at 0).
- The **actions** allowed in the environment are 4: *Move left*, *Move down*, *Move right*, and *Move up*.

Additional Exercise: Frozen Lake (pt. 2)



- In this environment, the standard reward function is discrete. A reward bonus of $+1$ is assigned when the agent reaches the goal position; no signals otherwise.
- This reward function does not provide any insight into how to solve the task, leading to poor performance. The exercise consists of (i) adapting the algorithm to the given environment, and (ii) finding a good reward function for the task.
- A good proxy for the reward (i.e., the evaluation metrics) is the success rate. To obtain this value is sufficient to count the number of times the agent reaches the goal in a sequence of 10 episodes.