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:

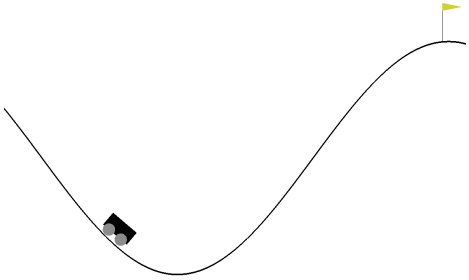
```
RL—Lab/lessons/lesson_11_code.py
```

Inside the file, you will find 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 done in order to adapt the algorithms to a different environment (e.g., network shape and hyperparameters).

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 for each timestep. 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:

```
# First observation: POSITION on the x-axis
# the position assumes values between -1.2 and 0.6
# the position of the goal is 0.5
position = observation[0]

# Second observation: VELOCITY
# the velocity assumes values between -0.07 and 0.7
velocity = observation[1]
```

Code Snippets

Following is a python-like code snippet to override the reward function:

```
# Get the observation before the update
previous_observation = np.array(self.env.state, dtype=np.float32)
# Extract the state information from the observation
position, velocity = observation[0], observation[1]
# Override the reward function (you can also use the action)
reward = None
```

Remember to 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 suggestion is to visualize the learning process:

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

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.

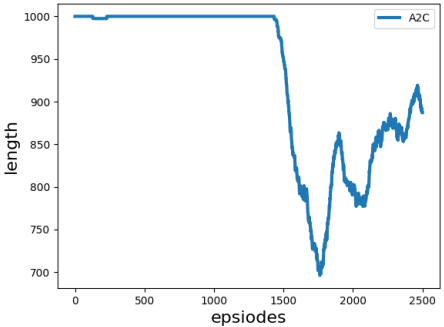


Figure: Note that obtaining this result requires time. You can stop the training as soon as the task is solved (i.e., first length below 1000 steps).

Additional Exercise: Frozen Lake

- 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*.