Reinforcement Learning Lab

Lesson 11: DRL in Practice

Luca Marzari and Alberto Castellini

University of Verona
email: luca.marzari@univr.it

Academic Year 2023-24

Today Assignment

Today's lesson will face two classical DRL problems: MountainCar and Mapless navigation. In particular, the file to complete the *MountainCar* task is:

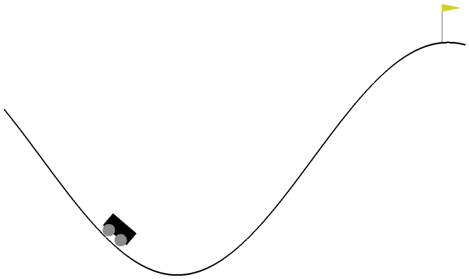
```
RL-Lab/lessons/lesson_11_code.py
```

We will also use the **ISLa playground** framework to train a mobile robot for the *mapless navigation* task.

Let's focus on the first assignment. Inside the file *lesson_11_code.py*, 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)**

Environment: MountainCar



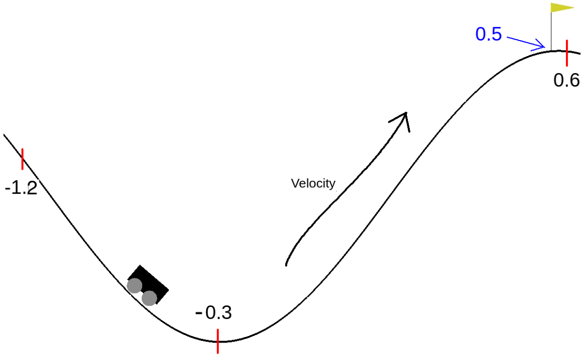
- The Mountain Car problem is a classic reinforcement learning problem with the goal of training an agent to reach the top of a hill by controlling a car's acceleration. The problem is challenging because the car is underpowered and cannot reach the goal by simply driving straight up.
- The **state** 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. Exploiting a continuous reward function that provides insightful information at each step is typically useful to achieve good performance.
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 first observation is the position of the car. The values range from -1.2 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 provided 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 and modify the returned values.
- Inside 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
```

A graphical visualization can be useful to better understand the environment, 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?

Note that the **training_loop()** function has been modified to directly return the mean episode length of the episode and not only the mean reward. 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.

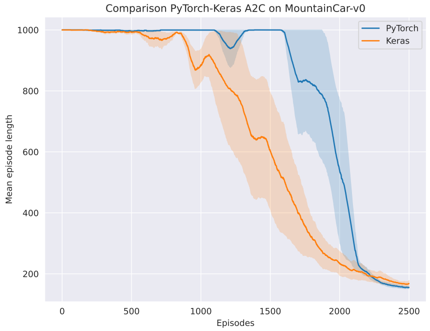
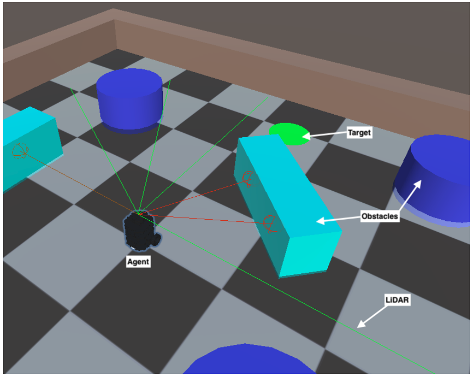


Figure: Note that obtaining this result requires time. You can stop the training as soon as the task is solved (i.e., length constantly below 1000 steps).

Mapless Navigation (pt. 2)



- A mobile robot has to control its motor velocities to reach goals that randomly spawn in an obstacle-occluded environment without having a map.
- The agent' state is composed of sparse lidar values sampled at a fixed angle and the goal's relative position (heading and distance) for a total of 9 features. Moreover, the agent controls its linear and angular velocity using 6 discrete actions.
- The **actions** allowed in the environment are: *Move forward*, *rotate in place left and right*, *move with fixed linear and angular velocity left and right*, and *no movement*.

Environment description:

Description	Values
Observation space (9,)	7 LiDAR values every 30° from [0°, 180°]. Each value is normalized between [0,1] 0 collision with obstacle, 1 Clear of Conflict
Action space (6,)	[[-45, 0], [0, 0], [45, 0], [-45, 0.05], [0, 0.05], [45, 0.05]] the first value is angular vel: grad/s the second one is the linear vel: m/s

The neural network will have 6 output nodes. During training (using a policy gradient method), the DNN will output a probability distribution for any action and randomly choose one based on those distributions (like in the Cartpole-v1 env).

```
# example
state = env.reset()
distribution = self.policy(torch.tensor(np.array([state])).type(torch.float)).detach().numpy()[0]
action = np.random.choice(self.env.action_space.n, p=distribution)
```

- *state* $= [x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ with x_7 distance from goal and x_8 heading.
- The first state value x_0 represents the ray at π(180 *degrees*), while the last one x_6 at 0 *degrees*.
- If you want to compute the distance from the goal between two consecutive timesteps, you can exploit *state[-2]* variable.

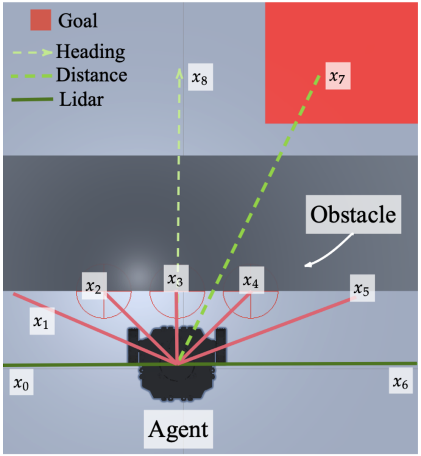


Figure: Observation space

Reward function

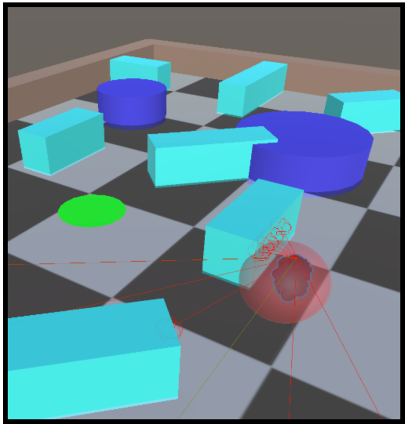
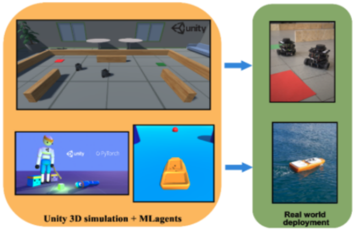


Figure: collision with an obstacle during mapless navigation

- In this environment, the standard reward function is sparse. A reward bonus of $+1$ is assigned when the agent reaches the goal position, -1 if a collision is detected and no signals otherwise.
- This reward function does not provide any insight into how to solve the task, leading to poor performance. As in the previous task you have to find the optimal reward function and hyperparameters to efficiently solve the task.
- A good proxy for evaluation metrics is the success rate. To obtain this value, it is sufficient to count the number of times the agent reaches the goal in a sequence of $1.5k$ episodes.

ISLa RL playground

For this part of the lesson, we will exploit the ISLa RL playground and Wandb service to plot the results!



https://github.com/Isla-lab/ISLa-RL-Playground

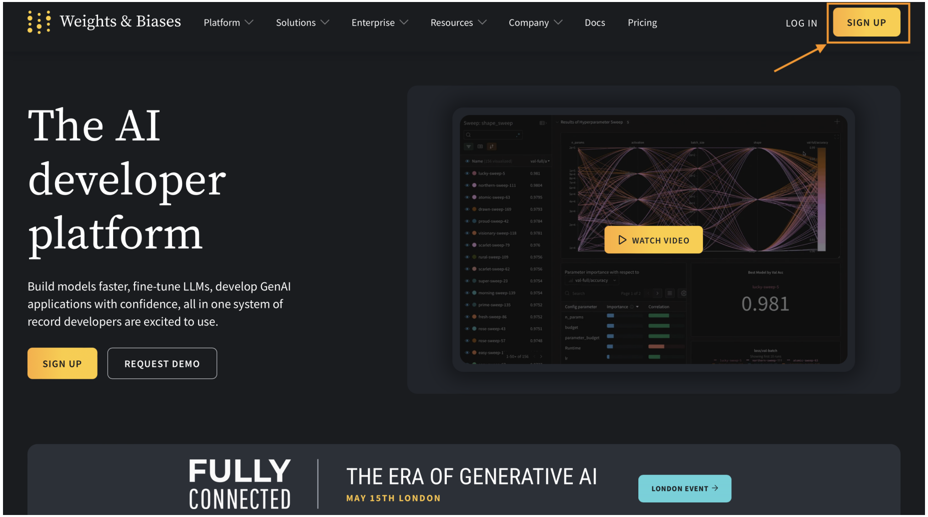
ISLa RL playground installation

- Clone the official ISLa playground repository here:
  git clone https://github.com/Isla-lab/ISLa-RL-Playground.git
- cd ISLa-RL-Playground/
- Create the environment and install the required packages:
  conda env create -f utils/isla-playground-env.yml
- Before running the scripts, remember to activate the environment:
  conda activate rl-playground
- create an account on wandb `https://wandb.ai/home`. Type: *wandb login* in the terminal and provide your API key when prompted.

Disclaimer

Create a new conda environment to avoid compatibility errors with the previous *rl-lab* env.

How to create a Wandb account?



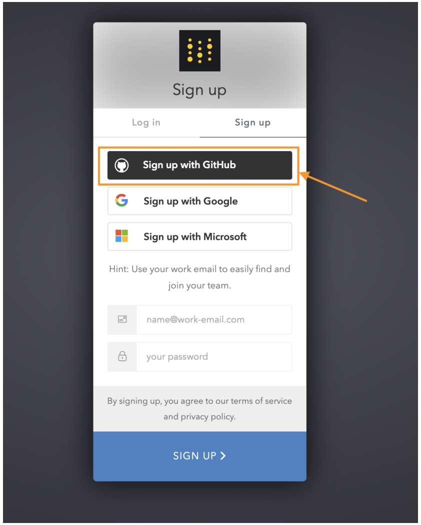
The AI developer platform

Build models faster, fine-tune LLMs, develop GenAI applications with confidence, all in one system of record developers are excited to use.

FULLY CONNECTED

THE ERA OF GENERATIVE AI

MAY 15TH LONDON



Hint: Use your work email to easily find and join your team.

By signing up, you agree to our terms of service and privacy policy.

- You can link your GitHub account.
- Once your account is created, log in, and you are ready to go.

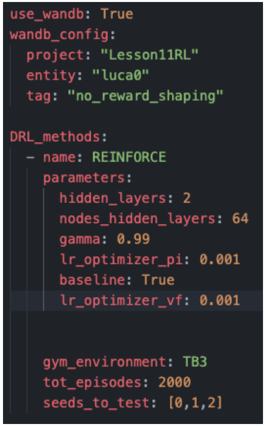
Reference

Here more information on wandb if necessary `https://docs.wandb.ai/quickstart`.

Config file and task to be done

The next step is to create the *config.yml* file and modify reinforce.py already provided in the folder *algos_template*. Once finished, *python main.py* in order to run the experiments

- Here, on the right, we report the general structure the file *config.yml* should have. In detail, the file is already available in the main directory of the repository; you have to change the project name, entity, and tag for the wandb part and the REINFORCE hyperparameters with your preferred ones.
- The exercise of this part consists of (i) completing the REINFORCE template. (ii) Override the reward function at **line 82 of the reinforce.py** file for the task. (iii) play with the hyper-parameters to discover the optimal ones to solve the task.



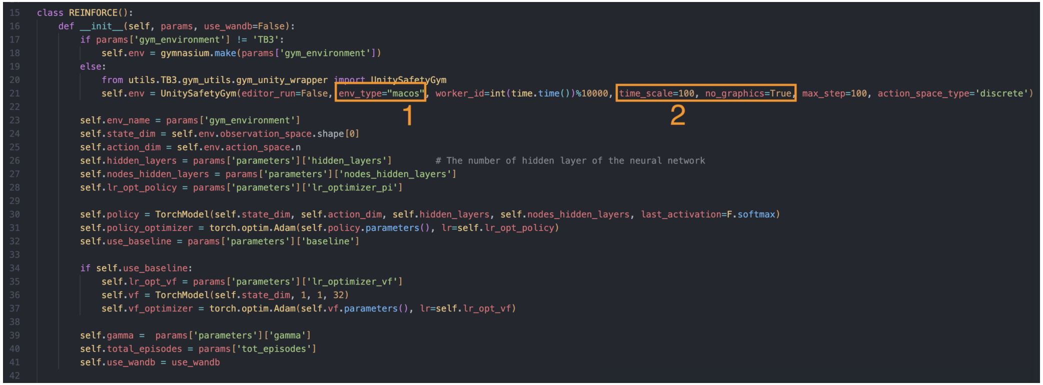
```yaml
use_wandb: True
wandb_config:
  project: "Lesson11RL"
  entity: "luca0"
  tag: "no_reward_shaping"

DRL_methods:
  - name: REINFORCE
    parameters:
      hidden_layers: 2
      nodes_hidden_layers: 64
      gamma: 0.99
      lr_optimizer_pi: 0.001
      baseline: True
      lr_optimizer_vf: 0.001


    gym_environment: TB3
    tot_episodes: 2000
    seeds_to_test: [0,1,2]
```

Env configuration and ML agents

This environment is based on Unity and MLagents. We provide a Gymnasium wrapper to interact with this environment as we normally have seen during all the laboratories.



```
15  class REINFORCE():
16      def __init__(self, params, use_wandb=False):
17          if params['gym_environment'] != 'TB3':
18              self.env = gymnasium.make(params['gym_environment'])
19          else:
20              from utils.TB3.gym_utils.gym_unity_wrapper import UnitySafetyGym
21              self.env = UnitySafetyGym(editor_run=False, env_type="macos", worker_id=int(time.time())%10000, time_scale=100, no_graphics=True, max_step=100, action_space_type='discrete')
22
23          self.env_name = params['gym_environment']
24          self.state_dim = self.env.observation_space.shape[0]
25          self.action_dim = self.env.action_space.n
26          self.hidden_layers = params['parameters']['hidden_layers']        # The number of hidden layer of the neural network
27          self.nodes_hidden_layers = params['parameters']['nodes_hidden_layers']
28          self.lr_opt_policy = params['parameters']['lr_optimizer_pi']
29
30          self.policy = TorchModel(self.state_dim, self.action_dim, self.hidden_layers, self.nodes_hidden_layers, last_activation=F.softmax)
31          self.policy_optimizer = torch.optim.Adam(self.policy.parameters(), lr=self.lr_opt_policy)
32          self.use_baseline = params['parameters']['baseline']
33
34          if self.use_baseline:
35              self.lr_opt_vf = params['parameters']['lr_optimizer_vf']
36              self.vf = TorchModel(self.state_dim, 1, 1, 32)
37              self.vf_optimizer = torch.optim.Adam(self.vf.parameters(), lr=self.lr_opt_vf)
38
39          self.gamma =  params['parameters']['gamma']
40          self.total_episodes = params['tot_episodes']
41          self.use_wandb = use_wandb
42
```

You have to change the line 21 with your os type in cell 1, and if you want to render the training, set *timescale=1* and *no_graphics=False*.

Expected Results

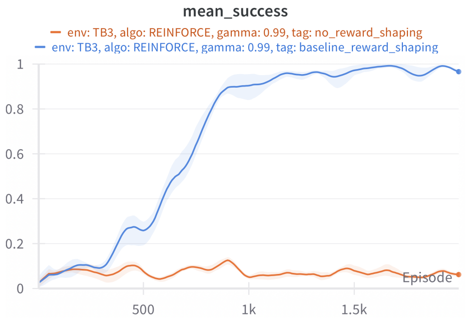


Figure: Comparison PyTorch REINFORCE in TB3 environment using 3 seeds [0,1,2] between reward shaping and not. x-axis reports the episodes, and y is the mean success rate.

Performance

Without any change in the reward function, the agent is unlikely to solve the task. A reward function that incentivizes the agent to reduce the distance from the goal at each timestep results in greatly improved performance.

Is it possible to improve further the performance?

Try to play and change different hyperparameters, like discount factor γ and the optimizer learning rate...

Different γs

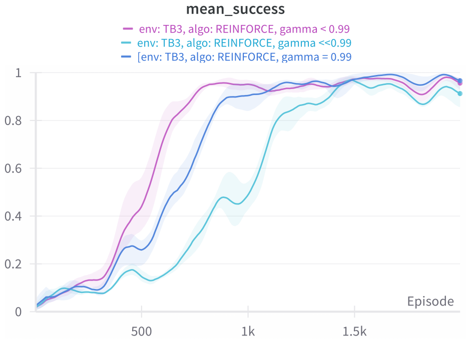


Figure: Comparison PyTorch REINFORCE in TB3 environment using 3 seeds [0,1,2] and different discount factor γ.

We can achieve different behaviors and sample efficiency for an efficient reward function, testing different gammas.

Can you guess the best γ?

Try to understand from this plot a possible reward function and discount factor that pushes the robot to reach the goal as fast as possible...