

# Exercise 3

## Gymnasium and PyTorch

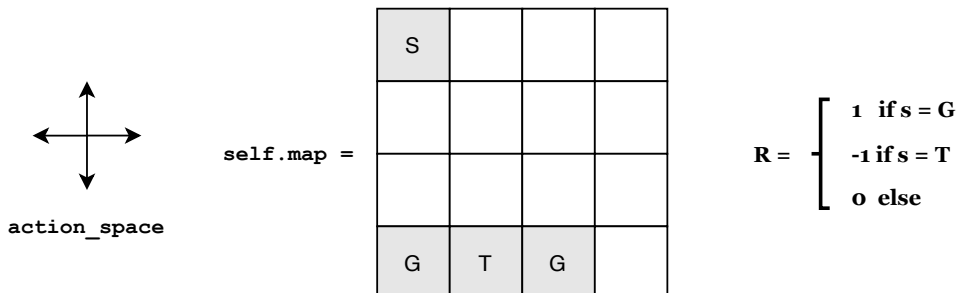
### 1 Gymnasium Gridworld

The Gymnasium/OpenAI Gym interface is a common standard for environments in reinforcement learning. It defines five methods that your environment has to implement. Providing the interface makes your environment compatible with nearly every RL framework (stable-baselines3, Tianshou, RLLib, ...).

The documentation and further information can be found here:

- <https://farama.org/Announcing-The-Farama-Foundation>
- <https://gymnasium.farama.org/>
- <https://github.com/Farama-Foundation/Gymnasium>

In this exercise, you will implement an environment based on the MDP definition. A simple grid world, similar to the last exercise, but with two different types of terminal states (two positives, **G**, and one negative **T**):



The agent starts in the cell marked **S**. At each time step, the agent can step up, right, down, or left if the agent reaches the goal (**G**), he receives a reward of 1, and the episode ends. If he falls into a trap (**T**), the episode ends with a reward of -1. Actions that would result in the agent leaving the board have no effect.

#### Programming Tasks:

1. `__init__()`. A `gymnasium` environment has to define its action and state space via two members, `action_space` and `observation_space`. Define the appropriate spaces inside the constructor. Also, add a member for the current environment state and assign the start state to it.
2. `reset()`. To reset the environment, a `reset()` method is required. This method should reset the environment state to a possible start state and return the observation for this state.
3. `step(action)`. To simulate a one-time step, the method `step(action)` is called with the agent's action. This method should simulate the environment dynamics and the agent's actions in one time step.

`step(action)` should simulate one step in the environment and return a tuple (`observation`, `reward`, `terminated`, `truncated`, `info`) that describes the transition. `observation` is the agent's perception of the new state. `reward` is the agent's reward in the new state. `terminal` is `True` if the next state is terminal, `False` otherwise. `truncated` can be neglected for exercise and can be set to `False`. `info` is a list or dictionary that contains additional (debug) information about the environment – for this implementation, you can return an empty dictionary (`{}`).

4. `render()`. This helper method visualizes the current state in a human- or agent-readable way. In our case, this method should print an ASCII representation of the current map and agent.
5. `close()`. This method frees up any resources (file handles, sockets, ...) that your environment needs to operate. In our case, the provided implementation (`pass`) does not need to be adapted.

Before continuing to the second exercise, we encourage you to test your implementation to find potential errors and problems. You can run the `env_test.py` file to execute random actions in your environment.

## 2 Setting up PyTorch

The deep learning framework PyTorch will form the basis of our implementations from this exercise onward. This week, we want you to get to know the framework on a simple supervised learning task. If you're already familiar with PyTorch, this exercise should be mostly familiar.

Install PyTorch to your local machine by following the instructions at <https://pytorch.org/get-started/locally/>. We recommend using Anaconda for the installation. You do not need to install CUDA, or use a GPU for this exercise.

Test your installation by creating a PyTorch `tensor` (equivalent to a numpy array) and printing it:

```
import torch
a = torch.arange(25)
a = a.reshape(5, 5)
print(a)
b = torch.tensor([1, 2, 3, 4, 5])

# Example 1
print(a * b)
# Example 2
print(a @ b)
```

What is the difference between examples 1 and 2?

## 3 Supervised Learning with Neural Networks in PyTorch

In this task, you will learn to train a simple classifier on the MNIST dataset. You can find documentation for the `torch.nn` module at <https://pytorch.org/docs/stable/nn.html>.

1. In the class 'Net', create a network with the following architecture:

- Conv 2D (32 Channels, 3x3 Kernel, Stride 1)
- ReLU
- Conv 2D (64 Channels, 3x3 Kernel, Stride 1)
- ReLU
- Max Pool 2D (2x2 Kernel)
- Fully Connected Layer (9216  $\implies$  128)
- ReLU
- Fully Connected Layer (128  $\implies$  10)
- Softmax

You'll need to create the appropriate members for your layers in `__init__` and perform the calculations in the `forward`-method. PyTorch will perform any needed gradient calculations for you.

2. Create an optimizer (e.g., `torch.optim.Adam`) for your model.
3. We already provide the basic outer training loop that will iterate over epochs and batches in `main`. Implement one step of prediction, loss computation (e.g. cross-entropy), gradient calculation, and optimization in the `train` and `test` methods.
4. Run the training. Your training and evaluation losses should decrease substantially.