Reinforcement Learning Assignment 2024 - 2025

Reinforcement Learning algorithms in OpenAI Gym and Minihack

Marianne Defresne, Giuseppe Marra

March 2025

1 Introduction

In this assignment, you will implement reinforcement learning models and algorithms that have been showed during the lectures. You will apply them in grid-world game-oriented environments.
You will be working with **Python**. The assignment is based on two frameworks:

- **OpenAI Gym**, a standard API for Reinforcement Learning (RL) environments, in the `gymnasium` package (`https://gymnasium.farama.org/api/env/`).
- **Minihack**, a sandbox framework for easily designing rich and diverse environments for RL, based on the game of NetHack and providing an OpenAI Gym interface (`https://minihack.readthedocs.io/en/latest/index.html`).

2 Approach

The assignment is individual. You must implement the required algorithms by yourself and you are not allowed to share code with your peers.
You are allowed to use standard Python libraries (e.g. for plots) but you must implement every RL-related algorithm.
Your assignment must adhere to general scientific standards of credit attribution. More details on this can be found at `https://www.kuleuven.be/english/education/plagiarism/`. Moreover, the KU Leuven guidelines on the use of Generative AI apply: `https://www.kuleuven.be/english/education/student/educational-tools/generative-artificial-intelligence`

3 Submission

Your submission should include two items:

1. A **report** in **.pdf** containing the output of the different tasks (**max 10 pages**, including images);
2. A **code package** in **.zip** format

4 Deadlines

Submit Report and Code on Toledo **Before June 1st, 23:59**

5 Score

The assignment will be scored on a **20 points** basis, jointly based on your report and code. The 20 points will then be rescaled to the **40%** (i.e. 8 points) of the overall score for the course. The remaining **60%** (i.e. 12 points) will be determined by the exam.

6 Installation

You can easily install the two frameworks on which your assignment depends using `pip`:

```
pip install minihack
```

We provide the following three python scripts:

- `commons.py` is a collection of utility functions;
- `minihack_envs.py` is a collection of Minihack environments that you will use in your assignment.
- `quickstart.py` is a script with small snippets of code for getting started with some basics `gym` and `Minihack` functions.

7 Tasks

Your report should discuss the following tasks (mention the task numbers). We will provide you with lightweight, often under-specified interfaces. **This code is supposed to be a starting point for you and not a constraint. You are allowed to change such interfaces to fit your needs (e.g. adding arguments to functions, adding functions, etc).** The adherence of your code to the provided interfaces will not be considered or scored. The interfaces are only intended to help you start thinking about the implementation.
Every task with an quantitative outcome (e.g. plot of return curves) or qualitative outcome (e.g. game rendering) **must be included in the report and commented.**

Task 1: Environments (5 points)

In this first Task, you will start by exploring the OpenAI Gym API, the Minihack framework and how simple agents interact with these enviroments.

Task 1.1 (3 points) In this subtask, you need:

1. Encode a variable size ($n \times m$) Grid-World goal-finding environment in OpenAI Gym, by implementing the interface the `gym.Env` class (ref. https://gymnasium.farama.org/api/env/). There are four actions possible: up, down, right and left. The agent starts in position $(0, 0)$ and the goal is in position $(n-1, m-1)$. The agent receives a negative reward of -1 for each action. The agent will not move if it performs an action that would bring it off the grid (but it is still rewarded a -1). To do this, implement the abstract methods of the class `gym.Env`, i.e. `reset()`, `step()` and `render()`. For the `gym.Env.render()` function, only a simple `str` representation is required, but you are free to choose whatever human readable rendering method you prefer.
2. Implement a `RandomAgent` class. You can find the interface in `commons.AbstractAgent`. A random agent executes any action with equal probability, i.e. uniform distribution.

3. Implement an `RLTask` class. You can find the interface in `commons.AbstractRLTask`. In this assignment, we refer with an RLTask as an object that encodes the interaction between an Agent and an Environment. In particular, in an RLTask, an agent can interact with an environment in two ways:

   - for a certain amount of episodes when we are interested in the evolution of the return (e.g. during learning). Extend `commons.AbstractRLTask.interact`. The function should return the list of average returns $\hat{G}_k$ for all episodes k. If G_k is the return at episode k, the average return $\hat{G}_k$ at episode k is:

     $$\hat{G}_k = \frac{1}{k+1} \sum_{i=0}^{k} G_i$$

     where k is a 0-based index. By using average returns, we avoid spikes in the curves due to the stochastic nature of the exploration.
   - for a single episode when we are interested in having a qualitative understanding of the agent behaviour (e.g. plotting an episode). Extend `commons.AbstractRLTask.visualize_episode` by rendering the single steps of an episode (i.e. `gym.env.render`)

   Use a 5×5 instance of the GridWorld environment that you implemented. Plot the evolution of the average return over an interaction of 10000 episodes by the `RandomAgent`. After that, visualize the first 10 time steps of an episode of the `RandomAgent`.

Task 1.2 (2 points) Consider the four Minihack environments in Figure 1. The action spaces only allow for movements in the cardinal directions: up, down, left and right, as the GridWorld you implemented. All the environments encode undiscounted tasks (i.e. $\gamma = 1$).

- `id = minihack_envs.EMPTY_ROOM`. It is a simple goal-finding-like environment, where the goal is to reach the downstairs. The agent gets -1 reward for each step.
- `id = minihack_envs.CLIFF`. It is based on the Cliff-environment of Sutton and Barto book (Chapter 6). The reward scheme is: 0 when reaching the goal, -1 for each step. Stepping on the lava gives -100 reward and teleports the agent to the initial state, without resetting the episode. The episode only ends when the goal is reached.
- `id = minihack_envs.ROOM_WITH_LAVA`. It is a slightly more complicated goal-finding environment. The reward scheme is the same as the `CLIFF`. Refer to https://nethackwiki.com/wiki/Des-file_format for more information about the elements on the environment.
- `id = minihack_envs.ROOM_WITH_MONSTER`. It is an environment identical to the `EMPTY_ROOM`, but now a monster walks around and can attack and kill the agent. The reward scheme is the same of the `CLIFF` environment: 0 for reaching the goal, -1 for each step, -100 for any death (which teleports the agent in the starting position without resetting the episode). When attacked, the amount of health points lost is randomly determined. While info about health points can be obtained from other representations of the state, they are not included in the default `"chars"` representation. You do not need to focus on health points. While this prevents the agent from learning very advanced behaviours, the agent can still learn to avoid monsters, or to kill them. In fact, Minihack provides a simple way of attacking agents even in this simple navigation environments: the agent can attack monsters by moving towards them when located in an adjacent grid cell. It is interesting to check, in Task 2, which strategy the agent will learn (avoiding monsters? killing them? others?).

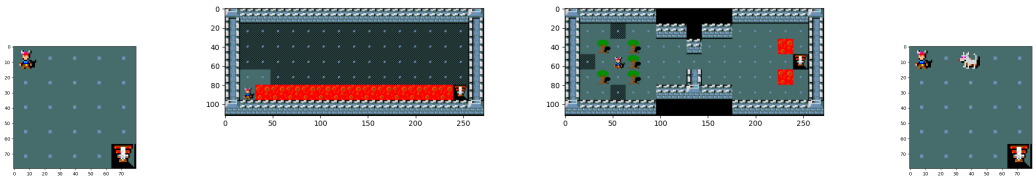

Figure 1: The four MiniHack environments

You can get an environment instance using `minihack_envs.get_env(id)`. See the `quickstart.py` script for some examples of use. You can use the `minihack_envs` as they are, but looking at the code can help you getting intuitions about how Minihack works and how you can build your own environments.
Now, design a `FixedAgent`, by extending the `commons.AbstractAgent` interface. A `FixedAgent` always goes down until it cannot move anymore, then it goes always right. Differently from the random agent, you now have to deal with the state of the environment (i.e. check whether the agent is by an edge of the lava). See Appendix for more info about Minihack states and for a link to the documentation.
Now, visualize 10 timesteps of an episode on the `EMPTY_ROOM` and `ROOM_WITH_LAVA` environments made by `FixedAgent` [1]. Of course, this is a dummy agent, only useful to check whether the interaction with environment is as expected. You will develop learning algorithms to solve the environments in the next set of tasks.

Task 2: Learning (10 points)

In this second Task, you will implement different reinforcement learning algorithms and you will compare them on different environments. We do not provide you with an interface for the learning, so you are free to choose how to encode learning Agents and, possibly, learning RLTasks. For example, you could leave the interaction agnostic to learning (i.e. do not change the RLTask) and make the Agent learning while acting (i.e. inside `Agent.act`). Or, you can make an RLTask which is aware of learning.

Task 2.1 (7 points) In this task you will implement three learning agents:

- Monte Carlo On-policy
- Temporal Difference On-policy (SARSA)
- Temporal Difference Off-policy (Q-learning)

All the agents explore using an ϵ-greedy policy during learning.
You will run an experimental campaign, where you will test the different agents on the four Minihack environments of Task 1.2. There are multiple comparison you can show. Examples include on-policy vs off-policy, MC vs TD, different learning rates, different epsilon for exploration, different stages of learning (i.e. number of episodes). Some environments could be better for one comparison than another. Investigate the settings and report the most interesting findings. Comparisons can be shown by plotting together different average return curves, like you did in Task 1.2. At least 4 comparisons should be included in your report but you are free to add more. This is a more scientific task. Keep a critical approach. For

[1] Notice that if you use the `Env.render` of the Minihack environment only a textual representation of the state is printed on the terminal. Following the `quickstart.py` snippets, you *can* use instead the "pixel" representation, as in Figure 1. In this case, you may need to slightly modify your previous RLTask to use the pixel representation instead of the default `Env.render`

example, do you observe the same results you would expect w.r.t. what we said during the lectures? Or if something different happens, why? The goal of this Task is to show us that you are understanding what the different algorithms are doing and why they behave in a particular way.

Task 2.2 (2 points) Have you noticed that, in the CLIFF environment, SARSA and Q-learning agents converge to different strategies? Explain why. Now, implement a decreasing scheduling of the ϵ during learning (i.e. decrease the value of ϵ after each episode). You can use a linear decay:

$$\epsilon_t = \epsilon_{\text{start}} - t\frac{(\epsilon_{\text{start}} - \epsilon_{\text{end}})}{\text{max\_num\_episodes}}$$

where ϵ_{start} is the initial value (usually very high), ϵ_{end} is the minimum value (e.g. 0.0 or a low value), t is the current episode index and max_num_episodes is the maximum number of episodes used in the decay. Plot a sample trajectory using the target policy at different stages of the learning (i.e. at different episodes t) and explain the observed behaviour. Remember, that in SARSA the target policy is still the ϵ-greedy policy, while in Q-learning, the target policy is always the greedy policy.

Task 2.3 (1 point) Implement a Dyna-Q agent, by adding a background planning strategy to your Q-learning agent. Compare the performance of the Dyna-Q agent and the Q-learning agent in two of the four Minihack environments.

Task 3: Deep Reinforcement Learning (5 points)

In this Task, you will implement a Deep Reinforcement Learning agent on two environements: `id = minihack_envs.EMPTY_ROOM` and `id = minihack_envs.ROOM_WITH_MULTIPLE_MONSTERS`, which is an extension of the previous environment with multiple random monsters spawning. For this task you do not need to implement the agent on your own. Instead, you can use `stable_baseline3`. To install it:

```
pip install stable_baselines3
```

You can follow the simple tutorials on https://stable-baselines3.readthedocs.io/en/master/guide/quickstart.html.

In this task, you need to:

1. Implement and learn in the two environments two deep reinforcement learning agents: *a)* a deep Q-learning agent (`stable_baselines3.DQN`) and *b)* an actor critic method (experiment freely with `stable_baselines3.A2C` or `stable_baselines3.PPO`). Refer to the course material and to the documentation of `stablebaseline3` to understand how to properly tune the algorithms.
2. Compare the two methods in terms of time of convergence and final reward.
3. Compare the table version and the deep version of the Q-learner and comment the advantages and disadvantages of the two techniques.

Different dimensions may require your attention in order to properly learn the deep reinforcement learning agent. First, the neural network used to encode states (e.g. MLP, CNN). Second the state representation: you can either use the `chars` representation (see below) or you may want to manually engineer some features to facilitate the learning (in alternative or in addition to the previous). Avoid using the pixel representation: they do not bring any additional information but will make interaction with the environment slower and learning harder. If you need to modify the representation of the

state you have multiple options. For example, you can modify the first layer of the policy or q-network (see https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html#custom-feature-extractor). Another, simpler, solution is to build a wrapper around your environment that modifies the state representation (see https://gymnasium.farama.org/api/wrappers/observation_wrappers/)

Minihack states

Minihack states are very rich, as you can see at https://minihack.readthedocs.io/en/latest/getting-started/observation_spaces.html and in the `quickstart.py`.
By default, `minihack_envs.get_env` provides only the `"chars"` representation, and the `"pixel"` representation when provided the argument `add_pixel = True`. Notice that adding pixels increases the interaction time due to the need to render the images. It is advisable to re-instantiate your environment (and possibly your RLTask) between longer episodes interactions (i.e. `AbstractRLTask.interact`) and short interactions for plotting (e.g. `AbstractRLTask.visualize_episode`). The `"pixel"` representation is mostly for visualization while the `"chars"` representation is the symbolic representation of a state you want to deal with.