Übungen zu “Deep Reinforcement Learning”

Wintersemester 2024 Zettel 2

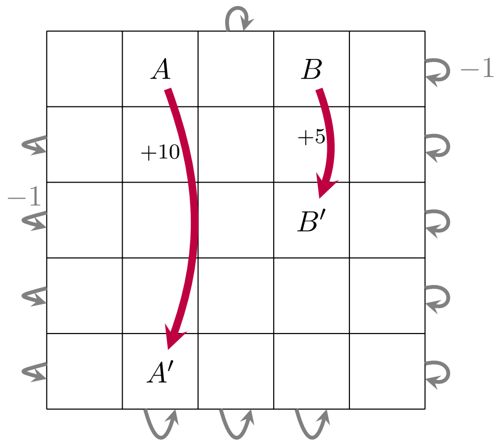
Ausgabe: 21.10.2024, **Abgabe:** 03.11.2024, **Besprechung:** 06.11.2024

Information - Some tasks use the Python library Gymnasium (Link). In those tasks, you will find instructions to help familiarize yourself with the library. Additionally, you can find an example of how to create and wrap environments under this Link. If you have further questions, please contact your exercise instructor Janosch Bajorath. The questions will be collected and discussed during an online meeting on the 30.10.2024. An announcement will follow.

Aufgabe 2.1 Value Function for a Grid-World Environment: (3 points = 2 + 1)

In this task, you will learn how to apply the Bellman equations to compute the State Value Function and Action Value Function in a given Grid-World environment. You have already been introduced to the Grid-World environment depicted below in the lecture, which represents a finite Markov Decision Process (MDP).

Description of the environment:



- **State Space** - The Grid-World is divided into fields, each field representing a state.
- **Action Space** - The agent can perform four actions in each state: North, South, East, or West.
- **Environment Dynamics** - Movements are deterministic and lead to the adjacent field in the chosen direction. Attempting to leave the grid leaves the agent’s position unchanged.
- **Reward Structure** - The agent receives no reward (0) for moving along the grid. Attempts to leave the grid result in a penalty of (-1). In the special states (A) and (B), the agent receives a reward of (+10) and (+5) respectively, and is moved to (A’) and (B’).

Subtasks 2.1:

(a) **Bellman with State Value Function** - Create a Python program that represents the Grid-World environment along with the possible actions. Develop an iterative algorithm that uses the Bellman equation to compute the State Value Function for all states. Use a random policy as the strategy and a discount factor of $\gamma = 0.9$. Visualize the convergence behavior of your algorithm through appropriate graphs and describe the results of your calculations. Pay particular attention to explaining how using a random policy affects the values of the State Value Function.

(b) **Bellman with Action Value Function** - Instead of the State Value Function, now determine the Action Value Function, again using a random policy. Examine the convergence behavior of the Value Function and graphically represent your observations to clearly illustrate the convergence pattern. Compare this approach with that of the State Value Function and explain possible differences and similarities between the two methods in writing.

Aufgabe 2.2 OpenAI Gym: Implementing the Grid-World: (3 points = 2 + 1)

In the following exercises, we will continuously work with environments from the Gymnasium library, which is why you will now implement your Grid-World environment as a Gymnasium environment. First, get an overview of the Gymnasium API and its features by reading the related paper (Link).

(a) **Creating the Grid-World Environment** - Implement your Grid-World from the previous task as an Environment for the Gymnasium library (Link). Create a Gymnasium environment by declaring a new class inheriting from the 'gymnasium.Env' class (GitHub-Link). Transfer your Python implemented Grid-World into the newly created class by implementing the following methods:

- **__init__()** $\rightarrow$ `None` - Initializes the environment, defines the action and observation spaces, as well as other environment parameters. Ensure that the action space (self.action_space) and the observation space (self.observation_space) are correctly defined. These specify what actions are possible and how states are represented.
- **reset()** $\rightarrow$ `observation` - Resets the environment to the initial state and prepares it for a new episode. Ensure that all necessary internal variables are reset, such as self.done and the state of the environment. The initial observation state of the environment should be returned.
- **step(action)** $\rightarrow$ `(observation, reward, done, truncated, info)` - Takes an action, performs it in the environment, updates the environment's state, calculates the reward received, and informs if the episode is over. Ensure that the action is within the defined action space. Update the state and calculate the reward. Check and update if the episode (self.done) is over and if it was truncated for other reasons. The individual return values represent (state of the environment after action, numerical reward, Boolean if the episode is over, Boolean if the episode was truncated, dictionary with additional information).
- **render(mode)** $\rightarrow$ `Optional[np.ndarray]` - Visualizes the current state of the environment. Depending on the mode, this can be done through text output, graphics, or other representations. The mode parameter ensures that different display types are supported. Depending on the mode, this method may not require a return value (e.g., 'human') or may provide an image array (e.g., 'rgb_array').
- **close()** $\rightarrow$ `None` - Closes all resources the environment may have opened, like graphics windows or files. This method only needs to be implemented if resource management is necessary; otherwise, it remains empty.

Please consider the following note:
Defining action and observation spaces is crucial for the agent's interaction with the environment. These must be clearly defined and follow the Gymnasium library standard. An overview of possible spaces can be found here. The flags `done` and `truncated` are used to distinguish between normal episode endings, like goal achievement, and early terminations, such as time limits. Additionally, it is advisable to use the `info` dictionary to provide supplemental information that can be useful for debugging, reinforcement learning algorithms, or evaluation.

(b) **Testing the Environment** - The next step is to register the environment (Link). To do this, use the register function from the module `gymnasium.envs.registration`. Then, you can test the registered environment. Instantiate the environment and create a loop that simulates an episode. Perform actions by randomly selecting actions from the action space using '`action_space.sample()`' and create images (e.g., screenshots) depicting 4 successive steps of the environment.

Aufgabe 2.3 Controller for the CartPole Gymnasium Environment: (4 points = 1 + 1 + 2)

The goal of this task is to develop a controller that manages to keep the pole upright for at least 200 time steps. Both an intuitive and a reinforcement learning-based approach should be implemented to solve the problem.

The CartPole environment is available as 'CartPole-v1' in the Python library Gymnasium (Information about the environment can be found here).

(a) **Conceptualization of the RL Problem** - Describe the CartPole environment as a Markov Decision Process (MDP) and create a simulation loop that visualizes the CartPole environment to better understand its behavior (you can find help under the following (Link)). In your description, specify the state space, the action space, the environment dynamics, and the reward structure.

(b) **Intuitive Approach** - Develop your own policy for the CartPole environment based on observations of the environment (e.g., the tilt angle of the pole) without using reinforcement learning methods. For this, implement a function that represents a policy, i.e., selects a reasonable action based on the current observation. Improve your policy iteratively by incorporating additional state information (e.g., speed). Simulate your different policies over 200 episodes and create plots showing the average reward over several runs. Evaluate the effectiveness of your policy, discuss difficulties encountered, and analyze the performance of your policy in terms of stability and goal achievement.

(c) **RL Approach** - The CartPole environment uses continuous observations. Implement an observation wrapper in Gymnasium that converts the continuous state space of the CartPole environment into discrete states (information on Gymnasium wrappers can be found here). These discrete states can now be considered individual decision points in the MDP, enabling the use of an RL approach using a lookup table. Develop an approach where, for each discrete state, an independent bandit is trained to find the best action in each state. Use an ϵ-greedy learning algorithm for decision-making in each discrete state. Visualize the learning progress with plots showing the reward over time for the individual bandits. Discuss how well this approach contributes to stabilizing the pole (performance and convergence behavior of the algorithm) and reflect on the efficiency of discretization in terms of the model's stability and learning capability.