Probabilistic Artificial Intelligence Task 4: Reinforcement Learning

1 Task Description

Your task is to develop and train a Reinforcement learning agent which will keep a pole mounted on a cart in an upright position Figure 1. To control the pole, the agent has a motor that can apply torques u in range of $[-1, 1]$, i.e., $u \in [-1, 1]$. You will implement an off-policy RL algorithm, such as TD3 or MPO, to solve this control task. Furthermore, we provide you with a simulator of the cartpole, with which your RL agent can interact and learn a control policy for the task.

This '.pdf' is meant to give a task-specific information. For logistics on the submission and setup, see the description on the task webpage. In this document, we will first explain the task environment and scoring. Secondly, we will outline the code structure provided in the code template.

2 Environment and Scoring Details

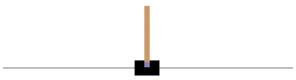

Figure 1: cartpole from the OpenAI gym classic control suite.

At each discrete time step, the controller can control the cartpole by applying a one-dimensional horizontal force to the cart, thus pushing it left or right. The simulation expects an action in the range $[-1, 1]$. The state is 4-dimensional and contains the position and velocity of the cart, as well as the angular position and angular velocity of the pole. The goal is to accumulate as much reward as possible in 200 timesteps. During learning, after 200 timesteps the episode ends and the environment is reset to a random initial state. As the task consists of balancing the pole, the reward at each time step is simply 0 if the pole is upright, and -1 otherwise. As the pole is initialized in an upright position, the return of a random policy is on average less than -180. On the other hand, the optimal policy would achieve a return of 0.

Since the focus of this task is the implementation of reinforcement learning algorithms, it is not necessary to have a detailed understanding of the cartpole environment beyond the observation and action space sizes (which can be found as attributes of the Env class as shown in 'solution.py'). However, if you are interested you can read more about the environment here.

When you run 'solution.py' your algorithm will have access to the standard cartpole environment that is commonly used in evaluating RL algorithms. To run 'solution.py' you will need to install the packages listed in 'requirements.txt'. You are encouraged to run 'solution.py' for testing.

Each single run of 'runner.sh' consists of a public and private evaluation. Each evaluation is repeated $N = 10$ times, each with a different random seed, which is used for initializing the environment, as well as the learning algorithm. For each seed, the algorithm is trained for 50 episodes in order to learn a policy. Each individual episode will contain exactly 200 transitions, for a total of 10k transitions. The score for each random seed is then estimated as the mean return of the learned policy over 300 episodes after training:

$$S = \frac{1}{300} \sum_{i \in \{1,\ldots,300\}} \sum_{t=1}^{T} R_t^i.$$

where R_t^i is the reward obtained at the timestep t, episode i of the evaluation, when using your final policy. The score S is then averaged across N seeds to provide the public score. The private score is calculated in the same way, but with a different set of random seeds. It is thus crucial to ensure that the algorithm does not overfit to the public evaluation.

Your goal is to maximize the final score and beat baselines. The public baseline is -172.5.

3 Solution Details

As stated in the task description, the code for this task is contained in 2 files: 'solution.py' containing template code and 'utils.py' where additional methods and functionalities for the task are stored. Classes *MLP*, *Actor*, *Critic*, and *Agent*, defined in the main solution file, are supposed to be filled out by you. Changes in other methods of the template file, like *main()* or any method in utils.py WILL NOT take any effect during evaluation.

The core of your solution should be implemented in the *Agent* class, more specifically, methods *train()* and *get_action()*, which will be called by our checker. The definition and arguments for the agent class constructor and two aforementioned methods MUST NOT be changed. Other methods in *Agent* or other classes are provided as additional structures that could guide you through your implementation. However, you are free to introduce your classes and methods or change the existing ones in any way, as that won't interfere with the final evaluation procedure. Still, we strongly recommend using our template classes and code for your implementation.

4 Hints

Most modern off-policy algorithms, such as MPO, can pass the baseline. Variance across random initializations is in general high in deep reinforcement learning. Make sure that your implementation choices result in improvements *on average* across a number of random seeds.