Reinforcement Learning
Exercise 1

September 26, 2021

1 Introduction

In this first exercise, we will take a first look at a reinforcement learning environment, its components and modify the reward function of a simple agent.

2 States, observations and reward functions

The provided Python script (`train.py`) instantiates a *Cartpole* environment and a RL agent that acts on it. The `agent.py` file contains the implementation of a simple reinforcement learning agent; for the sake of this exercise, you can assume it to be a black box (you don't need to understand how it works, although you are encouraged to study it in more detail). You don't have to edit that file to complete this exercise session - all changes should be done in `train.py`.

2.1 Cartpole

The *Cartpole* environment consists of a cart and a pole mounted on top of it, as shown in Figure 1. The cart can move either to the left or to the right. The goal is to balance the pole in a vertical position in order to prevent it from falling down. The cart should also stay within limited distance from the center (trying to move outside screen boundaries is considered a failure).

The state and the observation are four element vectors:

$$o = s = \begin{pmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{pmatrix}, \tag{1}$$

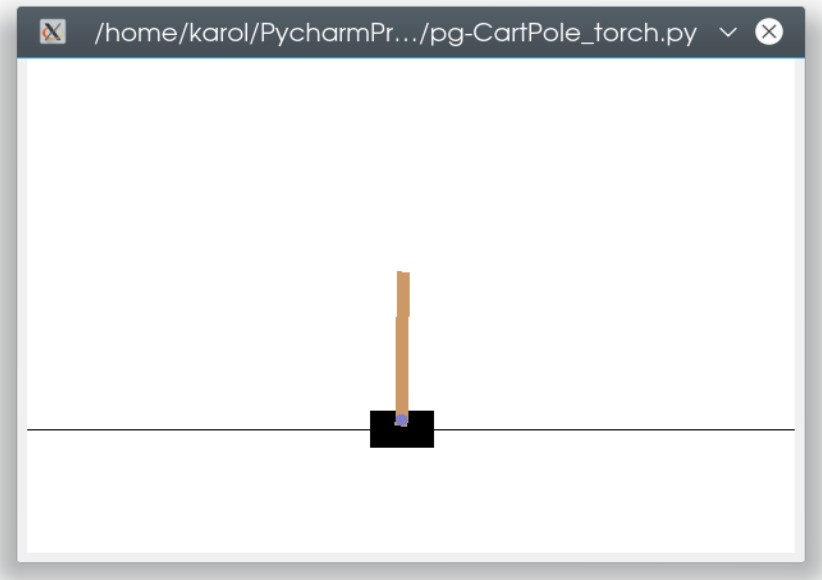

Figure 1: The Cartpole environment

where x is the position of the cart, $\dot{x}$ is its velocity, θ is the angle of the pole w.r.t. the vertical axis, and $\dot{\theta}$ is the angular velocity of the pole.

In the standard formulation, a reward of 1 is given for every timestep the pole remains balanced and the task is considered to be solved if the pole is balanced for 200 timesteps. Upon failing (the pole falls) or completing the task, an episode is finished.

2.2 Reward functions I

Run the `train.py` script. See how the cartpole learns to balance after training for some amount of episodes. You can use the `--render-training` argument to visualize the training (it will run slower when rendering). When the training is finished, the models are saved to `CartPole-v0_params.ai` and can be tested by passing `--test path/to/model/file.ai`. You can use the `--render-test` argument here.

Task 1 - 5 points Train a model with 200 timesteps per episode. Then test the model for 500 timesteps by changing line 136 in `train.py`. You can test the model with: `python train.py -t MODELFILE --render-test`

Note: the episode length and the number of episodes given by the command line argument `train--episodes` are two different things—the former controls how the pole will be trained to balance for during each attempt, while the latter sets the total number of such attempts during training. If you include plots in your submission, use the train episodes as the x-axis.

Question 1.1 — 15 points Can the same model, trained to balance for 200 timesteps, also balance the pole for 500 timesteps? Briefly justify your answer.

Figure 2: Variance in performance between multiple repetitions of the experiment.

Task 2 - 10 points Repeat the experiment a few times, each time training the model from scratch with 200 timesteps and testing it for 500 timesteps. Evaluate its performance by using the `--render-test` argument and remember the average test rewards.

Question 1.2 — 15 points: Are the behavior and performance of the trained model the same every time? Why/why not? Analyze the causes briefly.

2.3 Repeatibility

Figure 2 shows the mean and standard deviation throughout 100 independent training procedures. You can notice that there is a large variance between the runs of the script.

Question 2 — 10 points: Why is this the case? What are the implications of this stochasticity, when it comes to comparing reinforcement learning algorithms to each other? Please explain.

You can generate a similar plot by running the `multiple_trainings.py` script. If the script is slow, try disabling PyTorch multithreading (by running the script with `OMP_NUM_THREADS=1`; to do this, either export it in the shell with `export OMP_NUM_THREADS=1`, or place the variable before the command like this: `OMP_NUM_THREADS=1 python multiple_training.py`)

2.4 Reward functions II

Now we will focus on designing a reward function for a different environment, the *Reacher* environment, where a two-joint manipulator needs to reach a goal (see Figure 3).

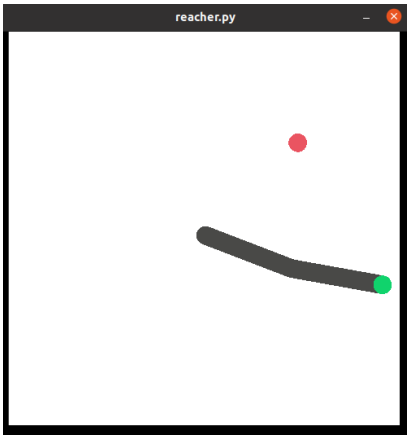

Figure 3: The Reacher environment

The Cartesian (x, y) position of the end-effector of the manipulator can be determined following the equation:

$$\begin{aligned} x &= L_1 \sin(\theta_0) + L2 \sin(\theta_0 + \theta_1), \\ y &= -L_1 \cos(\theta_0) - L2 \cos(\theta_0 + \theta_1), \end{aligned} \tag{2}$$

where $L_1 = 1$, $L_2 = 1$ are the lengths, and θ_0, θ_1 the joint angles of the first and second links respectively.

The state (and observation) in this environment is the two element vector

$$o = s = \begin{pmatrix} \theta_0 \\ \theta_1 \end{pmatrix}, \tag{3}$$

The action space now consists of 5 "options"; 4 correspond rotating the first/second joint left/right, and the final one performs no motion at all (the configuration doesn't change). The episode terminates when the agent reaches the target position, marked in red. To run the training script with the Reacher environment, use `python train.py --env Reacher-v0`.

Now, let us design a custom reward function and use it for training the RL agent (use the `new_reward` function for the implementation).

Task 3 — 20 points: Write a reward function to incentivise the agent to learn the following behaviors and include them in your report:

1. Reach the goal point located in $\mathbf{x} = [1.0, 1.0]$ (marked in red),
2. Keep the manipulator end-effector (the tip) moving at the maximum possible speed.

Train one model for each behavior with one reward function and include the model file and code in your submission.

Hint: Use the observation vector to get the quantities required to compute the new reward (such as the position of the manipulator). You can get the Cartesian position of the end-effector with `env.get_cartesian_pos(state)` and its Cartesian velocity with `env.get_ee_velocity(state, next_state)`. If you have trouble training your

model, try increasing the `--min-update-samples` and `--batch-size` values, and leave the training running for longer with the command line argument `--train-episodes-number`.

Task 4 — 10 points: Now, let us visualize the reward function for the first behavior (reaching the goal [1,1]). Plot the values of the first reward function from Task 3 and the learned best action as a function of the state (the joint positions). You can use the `plot_rew.py` script as a starting point. Make sure you include the plots in your report!

Question 3 Analyze the plots in Task 4.

Question 3.1 — 5 points: Where are the highest and lowest reward achieved?

Question 3.2 — 10 points: Did the policy learn to reach the goal from every possible state (manipulator configuration)? **Why/why not?**

3 Feedback

In order to help the staff of the course as well as the forthcoming students, it would be great if you could answer to the following questions in your submission:

- How much time did you spend solving this exercise?
- Did you find any of the particular tasks or questions difficult to solve?

4 Submitting

The deadline to submit the solutions through MyCourses is on Monday, 4.10 at 12:00pm (noon). Example solutions will be presented during exercise sessions that day (4.10).

Your submission should consist of (1) a **PDF report** containing **answers to the questions** asked in these instructions and **reward plots**, (2) **the code** with solutions used for the exercise and (3) the **trained model files** for each reward function (remember to report what reward functions were used). Please remember that not submitting a PDF report following the **Latex template** provided by us will lead to subtraction of points.

For more formatting guidelines and general tips please refer to the submission instructions file on MyCourses.

If you need help or clarification solving the exercises, you are welcome to come to the exercise sessions in weeks 37/38/39.