RWTH Aachen University
Institute for Data Science in Mechanical Engineering

Document Version 1.0.0, released 2022-06-24

Bonus Point Assignment II

Introduction

This document gives all necessary information concerning the second bonus point assignment for the class RLLBC in the summer semester 2022. This assignment consists of three Jupyter notebooks containing starter code for (A) the REINFORCE [1], (B) the Deep Q-Networks algorithm [2] and (C) the Proximal Policy Optimization algorithm [3]. OpenAI Gym [4] offers a clean environment interface and implementations of example environments often used in reinforcement learning. In this assignment, you will use some of them. As this is an assignment for bonus points, some questions may extend the scope of the lecture.

Formalia With this voluntary assignment, you can earn up to 5% of bonus points for the final exam. The bonus points will only be applied when passing the exam. A failing grade cannot be improved with bonus points. This assignment starts on 24.06.2022 and you have until 22.07.2022 23:59 to complete the tasks. The assignment has to be handed in in groups of 3 to 4 students. All group members have to register in the same group on Moodle under *Groups Assignment 2*.
Please hand in your solution as a single zip file including all three notebooks as well as the `solution` folder, which contains automatically generated export files. Do not clear the output of your submitted notebooks. Your solution has to be handed in via Moodle before the deadline.

Grading Your implementations will mostly be checked automatically via unit tests, so ensure that your notebooks run correctly from start to finish. Throughout the notebooks, there are short unit tests for many tasks that you can use to validate your implementation. However, note that these are not the unit tests that will be used to grade your assignment.

Jupyter To run the Jupyter Notebooks for this assignment, we recommend installing Jupyter locally, to be able to render the environment. Unfortunately we cannot give support in case your environment does not work. Alternatively, you can also use the RWTH JupyterHub. For this, search for the `[rllbc] Reinforcement Learning and Learning-based Control` profile and start the server. This profile was specifically created for this class and automatically comes with PyTorch and the other required dependencies. Once your server is started up, you can upload the notebooks and datasets and start completing the tasks.
To make the local installation as easy as possible we recommend to use the light-weight Miniconda Python distribution (or the larger version Anaconda if you want to do more extensive Python programming), which can be downloaded here. Installation instructions are provided here. We provide a

conda environment for this bonuspoint assignment. An overview (also useful as a cheat sheet) of the environment management system can be found here.
To install the environment provided in the file rllbc_bpa2_env.yaml and called rllbc_bpa2, simply navigate in your terminal (or console) to the file location and run

```
conda env create -f rllbc_bpa2_env.yaml
```

The environment can then be activated and deactivated with

```
conda activate rllbc_bpa2
conda deactivate
```

Rendering the environment is not supported on the JupytherHub. This means you can still use the JupyterHub to develop your code but then will not be able to render the environment. If you would like to see your environment, run the notebook locally on your computer.

Assignment Part A - REINFORCE

In this exercise you will program the REINFORCE algorithm [1]. You will train a REINFORCE agent to balance a cartpole in the upright position.
The CartPole environment [5] consists of a pole that is attached by an un-actuated joint to a cart that moves along a frictionless track. The pendulum starts upright. The goal is to keep the pole upright and prevent it from falling over by increasing or reducing the cart's velocity. The system is controlled by applying a force of $+1$ or -1 to the cart. A reward of $+1$ is provided for every timestep that the pole remains upright, including the termination step. The observations in the CartPole-v1 environment [4] we are using in this assignment are four dimensional and continuous:

Num	Observation	Min	Max
0	Cart Position (m)	-4.8	4.8
1	Cart Velocity ($\mathrm{m\,s^{-1}}$)	$-\infty$	∞
2	Pole Angle (rad)	-0.418 (-24 deg)	0.418 (24 deg)
3	Pole Velocity at tip ($\mathrm{s^{-1}}$)	$-\infty$	∞

At each timestep, one of two discrete actions can be taken:

Num	Action
0	Push cart to the left
1	Push cart to the right

For each timestep in an ongoing episode a reward of $+1$ is given. The episode ends when

- the pole is more than 12 degrees from vertical, or
- the cart position is more than 2.4 (center of the cart reaches the edge of or the display), or
- episode length is greater than 500.

Thus, the maximal accumulated reward for one episode is 500.

Figure 1: Two states from the CartPoleV1 environment. Left: Pole is in an upright position. Right: The angle of the pole to the vertical got too large and the episode is considered to be failed.

REINFORCE

a) The policy network $\pi(a|s) = P(A_t = a|S_t = s)$ maps states to action probabilities. Therefore the size of the input should match the size of the state space, and the size of the output should match the size of the action space.

Complete the implementations of the *__init__* and *forward* methods. You may choose activation, drop out, batch normalization, weight regularization, as it seems sensible to you.

Note: For this task, a small shallow network is sufficient (e.g., one hidden layer with 128 neurons).

b) The policy returns an action distribution for each given state. Implement the *sample_action* method, which samples one action from this distribution. *sample_action* returns both the action and the log-likelihood of choosing this action given the distribution.

Hint: The log-likelihood is important for the training later on, thus gradients must be able to flow through (i.e. don't use *.detach()*). You may use *torch.distributions.Categorical*.

c) Approximate the return of each state by computing the return over each trajectory.

$$G_t = r_t + \gamma G_{t+1}$$

Compute the expected return based on a series of observed rewards. Use *gamma* to discount future rewards.

To prevent large differences in the magnitude of the return, apply standardization to the expected returns (i.e., bring the mean to zero and the standard deviation to one). Otherwise, a longer trajectory will have a larger return, which can be detrimental to the training of the neural network.

d) Complete the implementation of the training loop.

- Get the action distribution for the current state from the policy. Use the *sample_action* method to sample an action from it and give the corresponding log-likelihood.
- Compute the policy loss *policy_loss*. After each episode, the policy π is trained to minimize

$$\mathcal{L} = \sum_t -\log \pi(a_t|s_t) G_t.$$

After training, your policy should be able to achieve an average reward of at least 487.5 over 100 episodes. You may tune hyperparameters and adjust your network architecture. However, do not increase the number of training episodes (though you may lower it, if you're able to successfully train with less).

Assignment Part B - Deep Q-Network

In this exercise you will implement a Deep Q-Network agent [2].

B1 - Replay Buffer

a) Deep Q-learning commonly uses replay buffers to reuse samples of old episodes and to break high correlation between similar trajectories and neighboring data points. This replay buffer will store $(s_t, a_t, r_t, s_{t+1}, d_t)$-tuples, denoting a transition from state s_t to s_{t+1} via the action a_t while observing the reward r_t. The variable d_t is a binary flag indicating whether the episode terminated with that transition.

Implement the following methods:

***add*:** Adds one transition to the buffer and replaces the oldest transition in memory. Compute the index where you want to store the new transition and store the data of the transition at the appropriate position. Remember to increase the *mem_cntr* after adding a transition.

Hint: Use modulo-division to compute the index.

***sample_batch*:** Samples one batch from the memory with the given *batch_size*. Return numpy arrays for *states*, *actions*, *rewards*, *next_states* and *is_terminal*.

Hint: You may use *np.random.choice* to choose random indices for the batch.

b) The buffer implemented in a) is initially empty. Fill the replay buffer with transitions sampled from the environment using random actions. Use a loop to generate and add transitions to the buffer until *buffer.is_filled()*. Reset the environment before each episode.

A transition is considered terminal if the pole fell, i.e. *done and env._elapsed_steps* $<$ *500*. Otherwise, reaching the time limit of 500 steps would incur a zero return later.

Random actions can be sampled via *env.action_space.sample()*.

B2 - Deep Q-Network

a) Create a network architecture for the Q-network. The Q-network models the function

$$s \rightarrow [Q(s,a) \quad \forall a \in \mathcal{A}]^T$$

where k is the size of the action space. Like before, choose a simple architecture with just a few layers. If required, you can always add layers later.

Hint: You can start with a similar architecture as in Task A.

b) Q-values are not an action distribution from which we could sample. Instead, we will use an ϵ-greedy strategy for action sampling to introduce randomness into the actions. ϵ-greedy chooses a random action with probability ϵ and uses the optimal action otherwise.

$$\pi_\epsilon(s) = \begin{cases} \text{random action } a & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases}$$

Implement the *epsilon_greedy* function.

c) Implement the loss function

$$\mathcal{L} = \sum_{(s_t, a_t, r_t, s_{t+1}) \in \Delta} \left(\underbrace{Q(s_t, a_t)}_{\text{Q-value}} - \left(\underbrace{r_t + \gamma \hat{Q}(s_{t+1}, a_{t+1})}_{\text{expected Q-value}} \right) \right)^2$$

where Δ is a batch of transition samples and $Q(s_t, a_t)$ are Q-estimates in *q_network* and $\hat{Q}(s_{t+1}, a_{t+1})$ in *target_network*. Essentially, the loss is the difference of the current Q-value and the expected Q-value.

- Compute the Q-values $Q(s_t, a_t)$ for the current state and action in the *q_network*.
- Compute the best actions

  $$a_{t+1} = \arg\max_a Q(s_{t+1}, a)$$

  under the *q_network* for the next states s_{t+1}.
- Compute the target Q-values $\hat{Q}(s_{t+1}, a_{t+1})$ in *target_network* for the next state and the corresponding best action.
- Zero-out all target Q-values for terminal states
- Compute the expected target Q-values

d) This training loop performs an update step on the Q-network every 4 simulation steps. Each update step samples a batch of transitions from the buffer, computes the loss and performs an update on the Q-net using the optimizer.

Every 2000 steps, the target Q-net and the current Q-net should be synchronized. Implement the synchronization step. In this, *target_network* is updated with the current weights of *q_network*. Make sure they don't accidentally share weights.

Finally, run the training. After training, your policy should be able to achieve an average reward of at least 487.5 over 100 episode. You may tune hyperparameters and adjust your network architecture. However, do not increase the number of training episodes (though you may lower it, if you're able to successfully train with less).

Assignment Part C - Proximal Policy Optimization

In this part of the assignment you will implement a Proximal Policy Optimization (PPO) [3] agent. PPO is a state-of-the-art policy gradient actor-critic algorithm. A policy (or actor) network for decision making as well as a value (or critic) network for advantage estimation are trained.
You will train a PPO agent to land a rocket on the moon in the LunarLander-v2 environment [4].
The state of the rocket is represented in an 8-dimensional observation that describes position, orientation and the respective velocities of the rocket as well as whether the landing feet of the rocket have contact to the moon's surface.
The landing pad the rocket is supposed to land in is indicated by two yellow flags and is always centered in the coordinates (0, 0). The goal of this environment is to reduce the distance to the landing pad quickly while maintaining low velocities and keeping the rocket upright. A positive reward is given, when feet have contact to the ground. A terminal reward is given depending on whether the rocket managed to land or not.

Num	Observation	Min	Max
0	Horizontal position (m)	$-\infty$	∞
1	Vertical Position (m)	$-\infty$	∞
2	Horizontal velocity ($\mathrm{m\,s^{-1}}$)	$-\infty$	∞
3	Vertical Velocity ($\mathrm{m\,s^{-1}}$)	$-\infty$	∞
4	Angle (rad)	$-\infty$	∞
5	Angular Velocity ($\mathrm{rad\,s^{-1}}$)	$-\infty$	∞
6	Leg 1 has contact to the ground	$-\infty$	∞
7	Leg 2 has contact to the ground	$-\infty$	∞

To land the rocket, the agent can fire one of the three engines of the rocket at each timestep or can do nothing instead. Firing the engines produces cost but fuel is unlimited.

Num	Action
0	No operation
1	Fire left engine
2	Fire main engine
3	Fire right engine

Proximal Policy Optimization

a) In the *update* function of the PPO class, compute the every-visit Monte-Carlo estimate of the state values given the data provided in the memory. The variable name of the MC-estimates is *discounted_reward*.

b) In the *update* function of the PPO class, implement the two surrogate loss functions of PPO that are used to define the overall objective in equation (7) of the PPO paper [3]

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right].$$

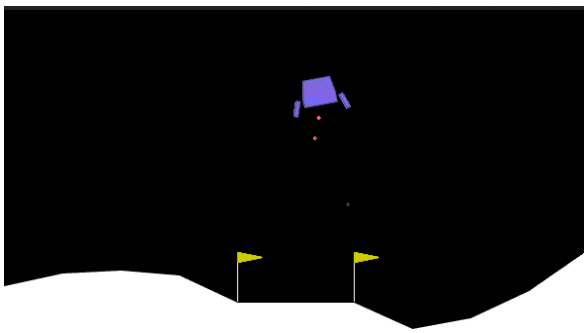

Figure 2: Land a rocket on the moon within the area of the landing pad indicated by two yellow flags.

c) In the training loop, implement the interaction between the old policy and the environment. Store state transitions and logarithmic probabilities in the replay memory.

d) In the training loop, implement the policy update. Use the *update* function of the PPO class. After updating the policy, clear the replay memory.

e) **(optional)** Play around with the hyperparameters of the algorithm. e.g. *eps_clip* and see how this influences the agent's performance.

After training, your policy should be able to achieve an average reward of at least 200 over 100 episodes. You may tune hyperparameters and adjust your network architecture. However, do not increase the number of training episodes.

References

[1] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Machine Learning, vol. 8, pp. 229–256, 1992.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," ArXiv, vol. abs/1312.5602, 2013.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," ArXiv, vol. abs/1707.06347, 2017.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[5] A. Barto, R. Sutton, and C. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, pp. 834–846, 1983.