RWTH Aachen University
Institute for Data Science in
Mechanical Engineering

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 2025. This assignment consists of two Jupyter notebooks containing code for (A) the Deep Q-Networks algorithm [1] and (B) the Proximal Policy Optimization algorithm [2]. OpenAI Gym [3] 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 beyond the scope of the lecture.

Formalia With this voluntary assignment, you can earn up to 4 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 11.06.2025, and you have until 16.07.2025 23:59 to complete the tasks. The assignment has to be handed in in groups of 2 to 4 students. All group members have to register in the same group on Moodle under *Assignment 2* until 13.07.2025 23:59.

Jupyter To run the Jupyter notebooks for this assignment, you can use the same virtual Python environment that you used for Bonus Point Assignment 1. If you did not complete the previous assignment, simply follow the instructions in the `student_guide.MD` file provided in the task folder to start your Jupyter Notebook.
The files will be made available to you on Moodle and must be uploaded after solving them. Please keep the directory tree of the assignment as it is and make sure that changes are saved regularly. Furthermore, make sure that group members always have the latest version available. Editing using collaboration tools such as "Google Colab" is possible in principle, but using them can lead to complications during grading. The same holds for using any IDE like Visual Studio Code. We therefore only offer support for the use of JupyterLab.

Grading The implementations will mostly be checked automatically via unit tests. For that reason, make sure that you do not delete any cells, do not rename functions or cell headers, and do not change the function's signatures. For debugging purposes, you are allowed to create new cells; however, their contents will be ignored during evaluation and thus should not include code vital to the functionality of the notebook. To make sure that grading works correctly, here are some essential guidelines:

- Please hand in your solution as a single zip file and stick to the directory tree in which the files were provided. Please only upload one zip file that includes the two task folders. Including your venv_student folder in the upload will result in exceeding the permitted file size limit.
- Ensure that in the logs folders for each task, only the folder containing your final submitted agent (including the agent_model.pt file) is included. Delete all other contents from the logs folders to reduce the submission size, as Moodle has an upload size limit
- Do not rename any of the files for the submission.
- Do not delete or add any cells in the notebooks.
- Stick to the class and function definitions given in the task description.
- Your solution must be submitted via Moodle before the deadline.

Logging The provided notebooks log interesting key metrics during the training run. To view those metrics you can either use the external tool ”Weights & Biases” (WandB) or Tensorboard. To use WandB you have to create a free account on their website. Although the completion of the tasks does not depend on the choice of logging, we recommend using WandB as Tensorboard offers less functionality and less insightful graphics.
For using WandB, set the *enable_wandb_logging* flag in the notebooks and enter your WandB API key as prompted once you start a training run.
For using Tensorboard, run

```
tensorboard --logdir path/to/logs --host localhost
```

in a console with the activated conda environment or alternatively, use the inline option found in the notebooks.

Video Rendering The notebooks provide the functionality to periodically render videos of the agent acting in the environment. If you want to make use of this functionality (by setting *capture_video* = *True*) you have to make sure to have **ffmpeg** installed. For installation instructions, see the official website.

Assignment Part A - Deep Q-learning

In this exercise, you will be working with the Deep Q-learning (or short DQN) algorithm [1]. You will train an agent to balance a cart pole in an upright position.
The CartPole environment [4] 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 time step that the pole remains upright, including the termination step. The observations in the CartPole-v1 environment [3] 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 time step, one of two discrete actions can be taken:

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

For each time step 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 exceeds 500.

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

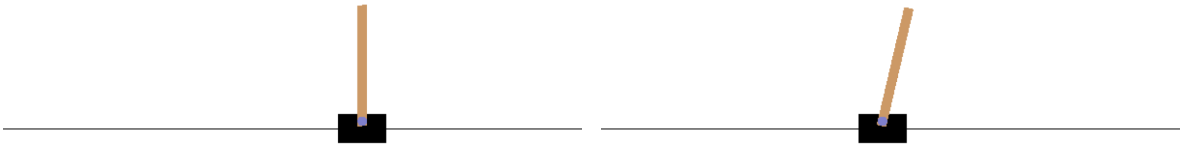

Figure 1: Two states from the CartPole_v1 environment. Left: Pole is in an upright position. Right: The angle of the pole to the vertical got too large and the episode is considered to be failed.

Deep Q-learning

Your good friend Tom has recently found interest in Deep Reinforcement Learning. He stumbled upon the Deep Q-learning algorithm and decided to implement it himself. Tom has always been a competent programmer but his poor knowledge and understanding of Reinforcement Learning algorithms has gotten the better of him - the implemented algorithm does not yield satisfying results! However, Tom is pleased to have you as his friend. Even more so now, as he is aware that you are currently attending a lecture on Reinforcement Learning at the University. He is confident that with your obtained expertise in the field (and your goodwill as a dear friend) it will be easy for you to get his implementation running.

Part a) - Fixing the Errors

You can find Tom's implementation in the notebook *dqn.ipynb*. In this part of the assignment, it becomes your task to fix Tom's mistakes and make the algorithm work. For that, you should take a look at his implementation of

- the algorithm-specific helper functions,
- and the training loop.

You should not modify the helper functions at */utils/helper_fns.py* or change the signatures of the functions in the notebook. Certain designated parts in the training loop also do not need any modifications. For debugging purposes, you can add new cells.

To receive full points in this part, you need to find and correct all of Tom's mistakes. In the end, you should be able to train an agent that receives an average reward of at least **490**. Note that a satisfying performance of the agent does not necessarily mean that you found all mistakes.

For the evaluation of your agent, you can make use of the evaluation infrastructure provided in the notebook. For insights in the training run, you can view key metrics logged by the notebook by either using Weights & Biases or plain Tensorboard.

For every training run, the notebook keeps track of the best-performing model and saves it to */logs/{run_name}/agent_model.pt*. Make sure to include your overall best model in the zip file you submit, renaming it to *dqn.pt*.

Hint: Tom is a decent programmer. The adjustments that are asked for are only very slight and pertain to the semantics of DQN. You never have to write multiple lines of code.

Part b) - Advancing DQN

During his first implementation attempt, Tom read something about the so-called "overestimation bias" that can happen in the vanilla version of DQN. He told you that to counteract this problem, there appears to be an improvement of DQN, called Double Q-learning or DDQN [5]. But looking more detailed into that as well was too much at once for him. However, maybe you can extend the favor of correcting his mistakes by implementing DDQN for him.

Complete the implementation of the method *compute_TD_target_DDQN*. This can be (and should be) done by using the provided functions of the Agent class. For this task, only change the code in the dedicated cell. If you find that DDQN improves your training, feel free to submit the model file obtained using DDQN instead of the one obtained in part a).

Assignment Part B - Proximal Policy Optimization

In this part of the assignment, you will work with a Proximal Policy Optimization (PPO) [2] 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 an agent to land a rocket on the moon in the LunarLander-v2 environment [3].

The state of the rocket is represented in an 8-dimensional observation that describes the 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 with 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 time step 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

Tom just loves you. Not only have you helped him find his mistakes in his Deep Q-learning implementation, you even implemented the DDQN extension in his stead. He was sitting there in front of

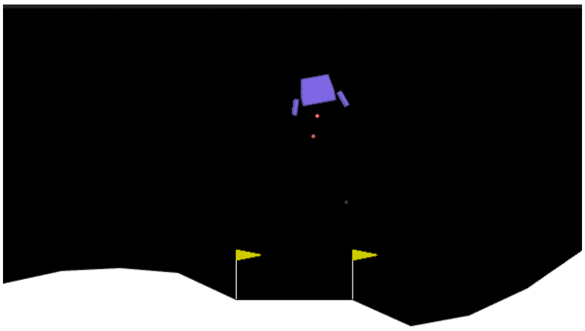

Figure 2: Land a rocket on the moon within the area of the landing pad indicated by two yellow flags.

his computer watching his little agent balance the cart pole over and over again. And he did so with a big smile on his face. Good work!
However, after some time playing around with different, more complicated environments he noticed that training a Deep Q-learning agent can be rather slow. Plagued by an inner feeling of dissatisfaction, he decided to dig deeper and found another promising Reinforcement Learning algorithm: Proximal Policy Optimization. The next time you see Tom, he is knocking on your door with a laptop in his hand and a worried look on his face. His PPO implementation is not working. You tell him that you are not familiar with the details of that algorithm and that you do not know whether you can help him this time, but it is no use. You are the smartest person he knows and the only one he can ask.

Part a) - Fixing the Errors

You can find Tom's implementation in the notebook *ppo.ipynb*. In this part of the assignment, it is your task to fix Tom's mistakes and make the algorithm work. For that, you should take a look at his implementation of

- the algorithm-specific helper functions,
- and the training loop.

You should not modify the helper functions at */utils/helper_fns.py* or change the signatures of the functions in the notebook. Certain designated parts in the training loop also do not need any modifications. For debugging purposes, you can add new cells.
To receive full points in this part, you need to find and correct all of Tom's mistakes. In the end, you should be able to train an agent that receives an average reward of at least **200**. You might find that you need to train for more episodes than in the previous task, and that you need the improvements that will be implemented in part b) to achieve this performance. Also note that during training, PPO follows a probabilistic policy to ensure exploration. This means that the achieved rewards during training might not reflect the true performance of the agent.
For the evaluation of your agent, you can make use of the evaluation infrastructure provided in the

notebook. For insights in the training run, you can view the key metrics logged by the notebook by either using Weights & Biases or plain Tensorboard.
For every training run, the notebook keeps track of the best-performing model and saves it to */logs/{run_name}/agent_model.pt*. Make sure to include your overall best model in the zip file you submit, renaming it to ppo.pt.

Hint: As in the previous task, the necessary modifications are only slight. If you find yourself in the need to write multiple lines of code, you are most likely on the wrong path.

Part b) - Refining PPO

During his research on PPO, Tom encountered several ways to refine the algorithm. He has already included some of them, like

- Annealing learning rate - slowly decreasing the learning rate as training progresses
- Gradient clipping - clipping gradients to a certain magnitude, preventing overly big gradients

In this part of the task, you are going to implement a few more. If you find that any of the refinements improves your training, feel free to submit the file of the model trained with the refinements instead of the one obtained in part a). Note that for each refinement, you should only change the code in the corresponding cell.

Normalization of Advantages To increase numerical stability while working with neural networks, it is common practice to normalize inputs and/or rewards. The first refinement you are going to implement is the normalization of the estimated advantages to zero-mean and unit variance for every minibatch. For a vector (of advantages) A, we achieve zero-mean and unit variance by the transformation

$$A' = \frac{A - \text{mean}(A)}{\text{std}(A)}.$$

Complete the implementation of the function *normalize_advantages*. Be careful that you never divide by a standard deviation equal to zero. You can prevent this by adding a minuscule bias to the denominator, e.g., 1e-8.

Value Function Loss Clipping It is possible to adapt the idea of a constraint policy update also to the value function approximator. In every update phase of the network, we prevent an overly large change in network parameters by clipping the value function loss to a constrained region around the old value function loss obtained during the rollout. More precisely, we compute the clipped value function loss for a minibatch of size n as

$$L^{CLIPVAL}(\theta) = \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\max\left((V_\theta(s_t) - G(s_t))^2, (V^{clip}(s_t) - G(s_t))^2\right),$$

where $V^{clip}(s_t)$ is the estimated value $V_\theta(s_t)$ clipped to an ϵ-region around the value estimated with the policy from the rollout $[V_{\theta_{old}}(s_t) - \epsilon, V_{\theta_{old}}(s_t) + \epsilon]$.

Complete the implementation of the function *compute_clipped_value_loss* in the notebook. For the hyperparameter ϵ use the same clipping coefficient as in the policy objective (*hypp.clip_coef*).

Generalized Advantage Estimate For the estimation of advantages, we can choose the number of steps after which we bootstrap as a hyperparameter. The formula for a general n-step advantage estimate is given by

$$\hat{A}_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{V}(s_{t+n}) - \hat{V}(s_t).$$

A larger number of steps decreases the bias of the estimation while increasing the variance and vice versa. Therefore, the number of steps manages the *bias-variance tradeoff* of the estimator. To get the better out of both worlds, it showed to be useful to combine the different n-step advantage estimates in the form of an exponential average, arriving at the so-called Generalized Advantage Estimate (GAE) [6]

$$\hat{A}_t^{GAE} = (1 - \lambda)\left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \ldots\right)$$

with λ being the coefficient for exponential decay. It can be shown that the expression can be written compactly, as

$$\hat{A}_t^{GAE} = \sum_{l=0}^{N-t} (\gamma\lambda)^l \delta_{t+l}$$

for an episode with time steps $0, 1, \ldots, N$ and with δ indicating the TD-error

$$\delta_t = \begin{cases} r_t - \hat{V}(s_t) & \text{, if } s_t \text{ terminal state} \\ r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) & \text{, else} \end{cases}$$

Furthermore, the generalized advantage estimates $\hat{A}_t^{GAE}$ can be computed efficiently for all time steps t as they fulfill the recursive formula

$$\hat{A}_t^{GAE} = \delta_t + \lambda\gamma \hat{A}_{t+1}^{GAE}.$$

Your task is to implement the generalized advantage estimation by completing the implementation of the function *compute_gae* in the notebook. Note that the rollout buffer might contain incomplete and/or multiple episodes that need to be handled correctly. You can test your implementation by using the provided unit test in the cell below. This is not the unit test that will be used to grade your submission.

Grading Each of the two tasks gives you up to 10 BPA points. The table below shows how points are distributed.

Points (BPA)	Bonus points (exam)
2	0.5
5	1.0
7	1.5
10	2.0
12	2.5
15	3.0
17	3.5
20	4.0

References

[1] 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.

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

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

[4] 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.

[5] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” 2015.

[6] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” 2018.