Deep Reinforcement Learning Assigment Report

Elias Ervelä
Student number: 2133501
E.Ervela@tilburguniversity.edu
GitHub link: https://github.com/ervelae/deep_reinforcement_learning_course

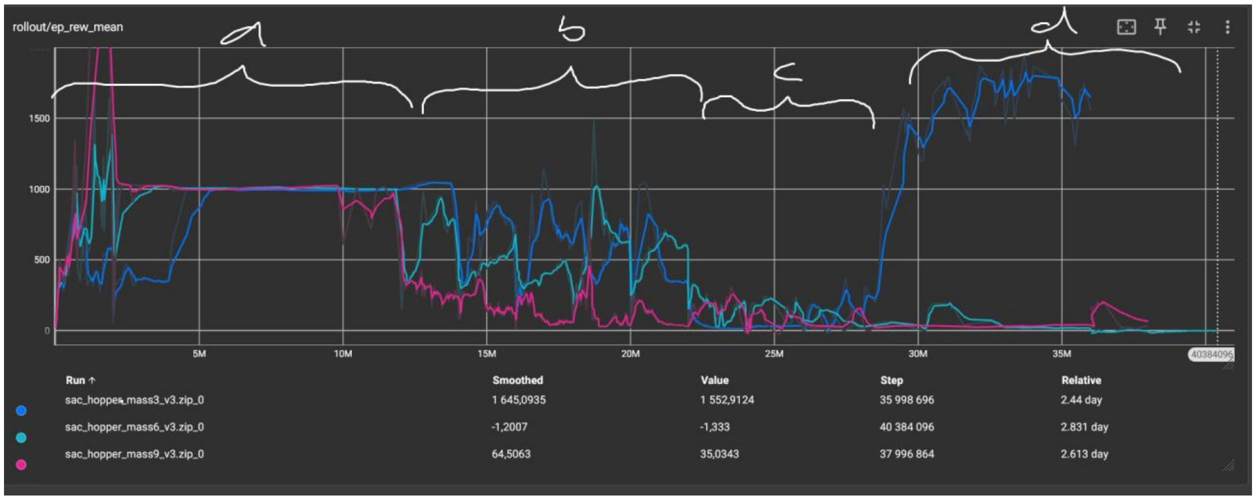
Part 1

In the Hopper environment we have a continuous "box" action space between [-1,1]. This means we can not use models that are only for discrete action space, e.g. DQN. I first started looking at DDPG for its simplicity, but ended up with trying first with TD3, since it is a more general version of DDPG. I wanted to start out with more simple methods, just to try how it goes.

I set up TD3 with the default parameters, but I added a little action noise of *sigma=0.1*. Since I am training only with my laptop CPU, to combat computational restrictions, I set *train_freq=(5, "step")*. This way each gradient decent updates are probably more accurate, and we do not have to update that frequently, especially without a proper GPU. However this means we do need more environment steps for the full training. The set *n_envs=4* of the multi-environment, since I have 4 cores in my laptop CPU.

The TD3 model plateaued at 1000 score. Looking at the simulation, the Hopper just stayed still, trying not to fall, and waited until the termination. I tried adding more and more action noise and in different environment seeds. I did not manage it to get out of that local maximum.

I switched the model to some other that might be better in exploration in the hopes that the Hopper starts hopping. Thus, I decided on SAC since it has a built in exploration in the loss function where it also maximizes the entropy. I set up the SAC with same parameters initially as TD3, with *train_freq=(5, "step")* and action noise of *sigma=0.1*. For every 2 million environment steps, I reset the environment with a new seed.



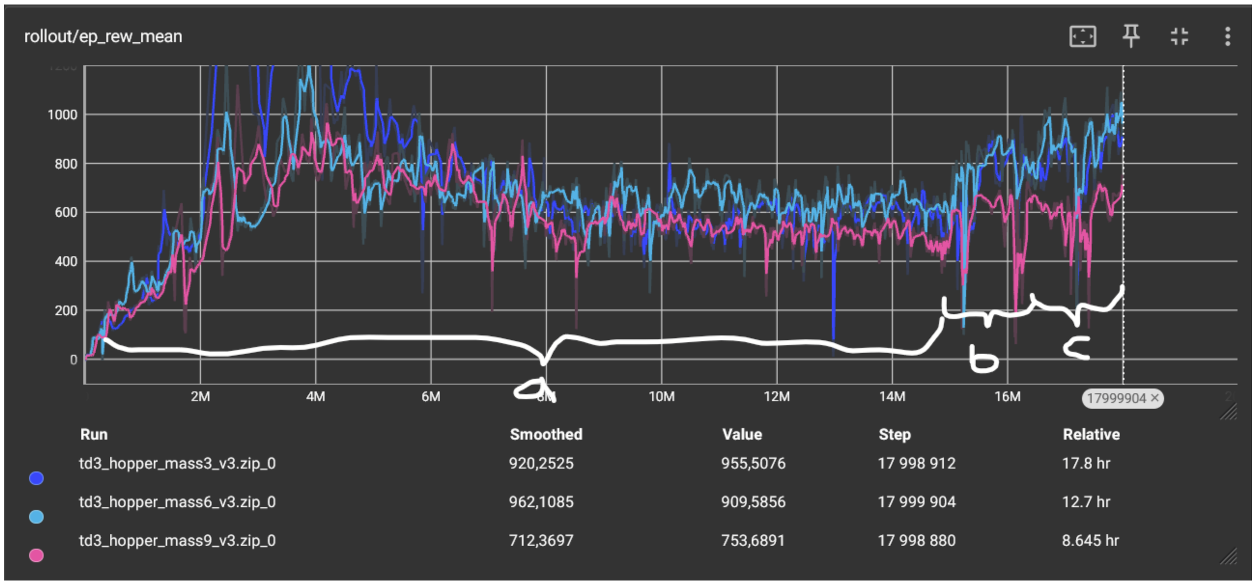
	Run ↑	Smoothed	Value	Step	Relative
●	sac_hopper_mass3_v3.zip_0	1 645,0935	1 552,9124	35 998 696	2.44 day
●	sac_hopper_mass6_v3.zip_0	-1,2007	-1,333	40 384 096	2.831 day
●	sac_hopper_mass9_v3.zip_0	64,5063	35,0343	37 996 864	2.613 day

Figure 1. Reward score during training of the SAC models for environments with *torso_mass* equal to 3 (blue), 6 (cyan), 9 (pink).

I had the same problem that all the models got stuck at the score of 1000 as can be seen in Figure 1 (a). Then I tired adding more and more action noise to get to do bigger movements, and increased *gamma* from *0.99* to *0.999* to value more the future rewards of going forward, and increased *learning_rate* from *0.0003* to *0.0006*, to get it out from the local maxima. That did not really help out either, as can be seen in Figure 1 (b). Then I added even more action noise with no avail as a can be seen in Figure 1 (c).

I decided to set the *healthy_reward* of the environment from *1* to *0.01*, to really encourage moving forward and not staying still, and lowering the *action_noise* down to reasonable levels. After that, the *torso_mass=3* model starter performing very good, as can be seen in Figure 1 (d)! But for some reason the *torso_mass= 6 and 9*, did not perform at all. I had set the action noise to *sigma=0.3*. My theory is, that the action noise was not enough for the higher mass hoppers to get hopping. I decided to reset the model weights to random with the *torso_mass= 6 and 9* and started retraining with *healthy_reward=0*.

At this point, I realized I had made one big mistake when training. Everytime I started re-training. e.g. with different seed, I did not load the latest corresponding *VecNormalize()* environment. Since the VecNormalize is taking the moving average during training to normalize the action space, it is changing during training, so we need to save and load the right VecNormalize environment to continue training.



	Run	Smoothed	Value	Step	Relative
●	td3_hopper_mass3_v3.zip_0	920,2525	955,5076	17 998 912	17.8 hr
●	td3_hopper_mass6_v3.zip_0	962,1085	909,5856	17 999 904	12.7 hr
●	td3_hopper_mass9_v3.zip_0	712,3697	753,6891	17 998 880	8.645 hr

Figure 2. Training of the TD3 model.

I returned to try TD3 model. The train process can be seen from Figure 2. A little bit frustrated at this point, I admit, I searched online for inspiraton for hyperparameters, and found this paper (https://arxiv.org/pdf/2005.05719). I followed those parameters mostly, but made some tweaks to them. I set the *n_envs = 32*, since I noticed the number of parallel environments was constraining my memory more than my CPU cores allowing me to gather interactions faster. Moreover, I set the action noise from *sigma=0.1* to *sigma=0.3,* since I have a lot of parallel environments, I can afford a more chaotic exploration. Also, I set *buffer_size* from *2e5* to *1e6,* since I have more chaotic exploration, so it is good to store more interactions, and I set my *batch_size* from *100* to *256* to stabilize the more chaotic exploration. I kept my neural network size as *[256, 256]*, and did not make it bigger or deeper, since I do not want to make the training too heavy.

From Figure 2 (a), we can see the first training run with different seed every 5e6 steps. It had plateaued clearly, so for the next part, I set a new seed and lowered the *action_noise* to *sigma=0.1*, which can be seen in Figure 2 (b). Then I lowered it to *sigma=0.05* for the next part, which can be seen in Figure 3 (c). At this point, I decided that this is good enough, and I do not have time to spend more time with this.

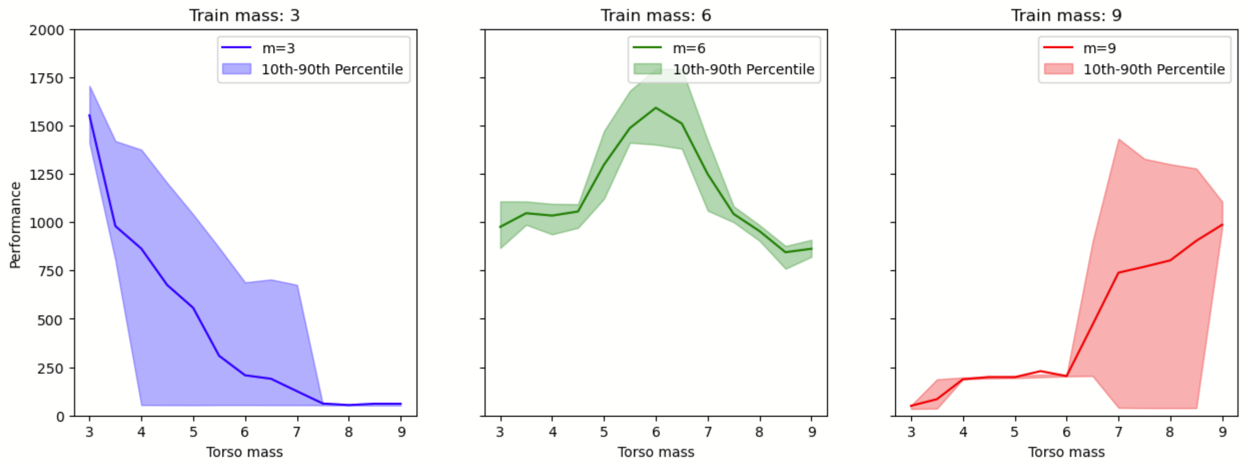


Figure 3. The results for evaluating with different masses with the TD3 model. For each mass from 3 to 9 with 0.5 increments, each one was evaluated with 100 episodes.

Evaluation results can be seen in Figure 3. We can see that the Train mass 9 -model, is performing a bit worse than the others as is expected when comparing them in Figure 2. In hindsight, I probably should have set the networks the same initialization seed, so that the models do not converge from each other too much when training for longer time. For some reason the Train mass 3 and 9 -models are much more chaotic. Also, I probably should have done a more systematic hyperparameter optimization from the beginning rather than trying to tune the parameters by hand one-by-one. However, the expected general shape is visible there in the figure, which I am very happy about.

Part 2

On-Policy method: PPO

For the on-policy model I went with **PPO**, due to it being one of the state-of-art methods and based on benchmark testings, is performing well on the bipedal walker. Hyperparameter optimization was done with a package called Optuna with the following parameter ranges:

```
    # Define the hyperparameter search space
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-3, log=True)
    n_steps = trial.suggest_categorical("n_steps", [1024, 2048, 4096])
    batch_size = trial.suggest_categorical("batch_size", [64, 128, 256])
    n_epochs = trial.suggest_int("n_epochs", 3, 12)
    gae_lambda = trial.suggest_float("gae_lambda", 0.8, 0.99)
    clip_range = trial.suggest_float("clip_range", 0.1, 0.3)
    vf_coef = trial.suggest_float("vf_coef", 0.5, 1.0)
    ent_coef = trial.suggest_float("ent_coef", 1e-8, 0.1, log=True)
    max_grad_norm = trial.suggest_float("max_grad_norm", 0.5, 1.0)
```

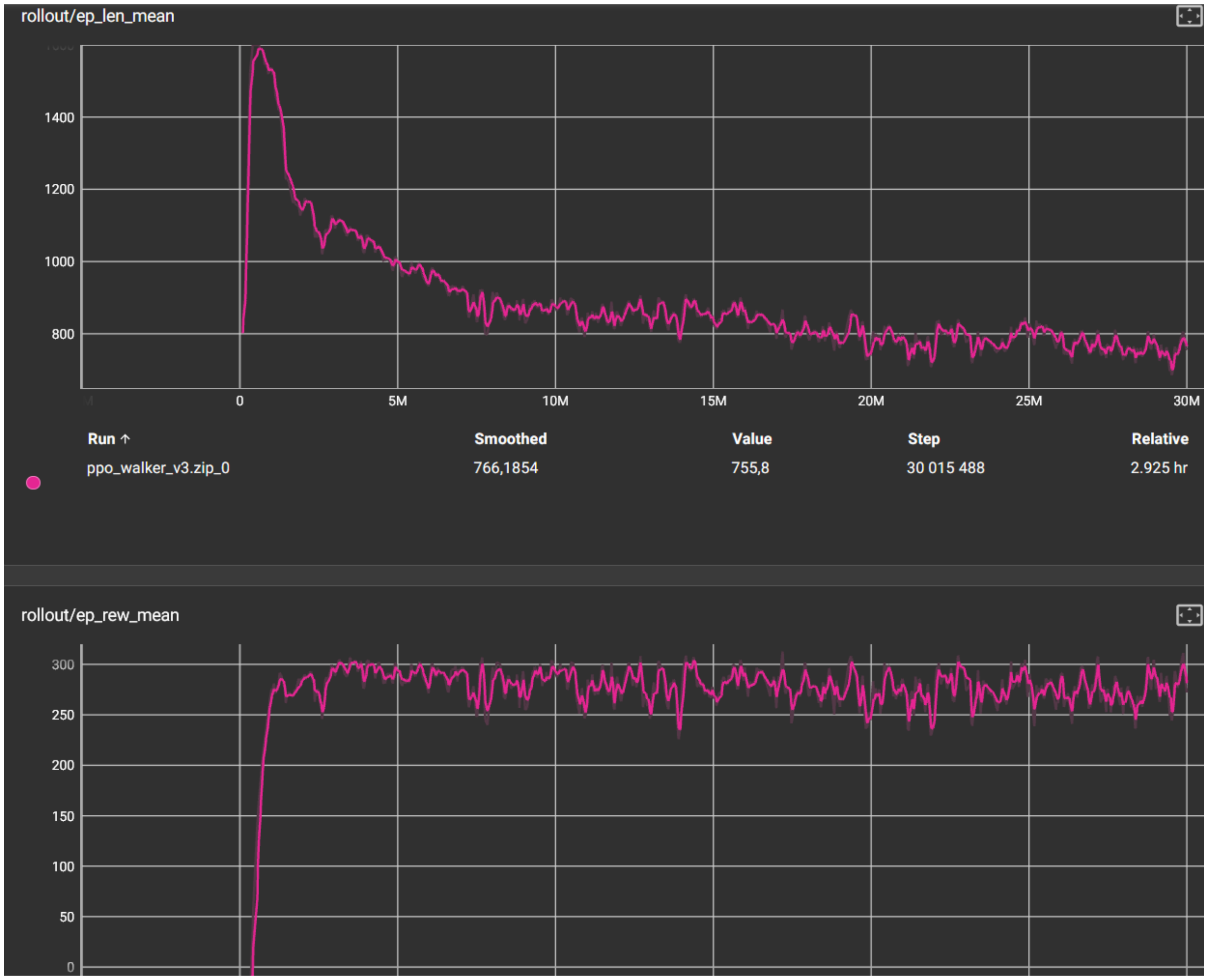
n_steps and batch_size, was chosen to be categorical here, since (n_envs * n_steps) needs to be a divisible by batch_size, so this is guaranteed with these categories. Learning rate and ent_coef was set to search logarithmic space, since there are multiple different magnitudes the optimal hyperparameters could be in. For other parameters, the search space is much narrower, so linear search was chosen.

Gamma was set to 0.999 for all hyperparameter optimization runs, because we want to maximize the whole run score, so higher is better. Sde noise was set to false, since from the initial testing, with that turned on, the model did not learn at all for some reason. Number of environments was set to 16, since that was the best my computer could handle with multiple parallel hyperparameter optimizations runs.

Each hyperparameter instance was trained 6e5 timesteps and evaluated on 10 episodes. The best score reached 275 with parameters:

```
[I 2024-06-06 22:06:27,816] Trial 25 finished with value: 275.4551026342516 and
parameters: {'learning_rate': 0.00031016290222399755, 'n_steps': 1024,
'batch_size': 128, 'n_epochs': 8, 'gae_lambda': 0.8610756390880515, 'clip_range':
0.2641177750386453, 'vf_coef': 0.9182477231951106, 'ent_coef': 4.9293649594261e-
08, 'max_grad_norm': 0.6337416874273284}. Best is trial 25 with value:
275.4551026342516.
```

Then the real model was trained with these parameters. It reached 300 score in around 3.1 million timesteps as can be seen in Figure 2 which took 18.12 min.



Run ↑	Smoothed	Value	Step	Relative
● ppo_walker_v3.zip_0	766,1854	755,8	30 015 488	2.925 hr

Figure 4. Training of PPO bipedal walker overnight. Score 300 was reach in around 3.1 million timesteps, which took 18.12 min. After that the walker learned to do it faster. In the end it learned to run properly switching which leg is in front.

Off-policy method: SAC

For the off-policy model, I went with SAC, due having somewhat familiarity of that method from the first part of the assignment.
Hyperparameter optimization was also done with Optuna, with the following parameter ranges:

```
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-3, log=True)
    buffer_size = trial.suggest_int("buffer_size", 100000, 1000000)
    batch_size = trial.suggest_categorical("batch_size", [64, 128, 256])
    tau = trial.suggest_float("tau", 0.004, 0.02)
    gamma = trial.suggest_float("gamma", 0.9, 0.999)
    train_freq = trial.suggest_categorical("train_freq", [1, 8, 16])
    gradient_steps = trial.suggest_int("gradient_steps", 1, 16)
    ent_coef = trial.suggest_categorical("ent_coef", ['auto', 0.1, 0.01, 0.001])
```

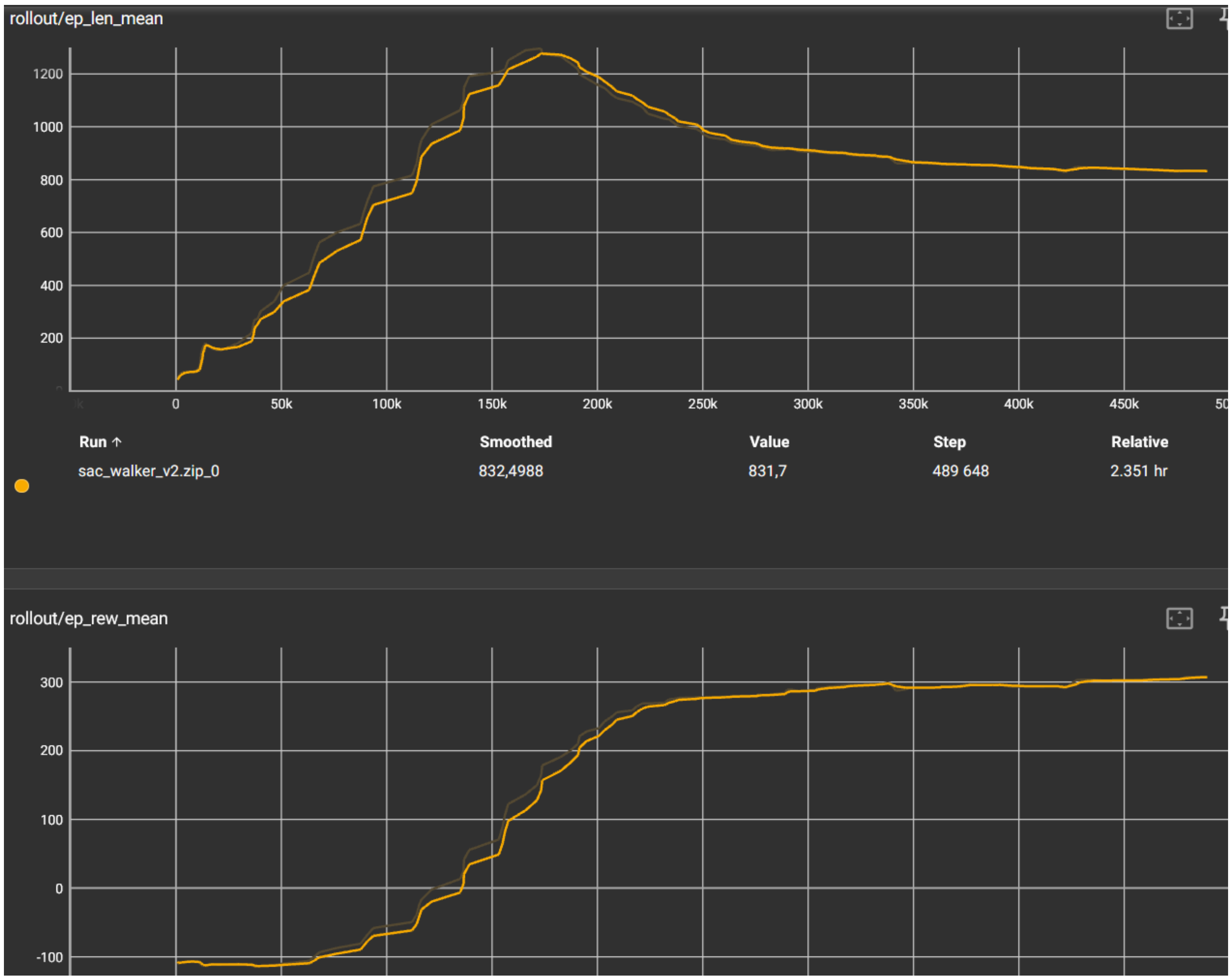
Batch_size and train_freq were initialized as categorized multiples of 2. Ent_coef was also put as categoized to include ‘auto’ as an option. Otherwise, the numerical parameters were initialized on a linear scale except learning_rate. I also tried here to optimize for the gamma parameter to see if that makes a difference.

For all optimization runs, I set the start of learning after 10000 timesteps, and I turned sde noise to True, since based on benchmarks I found, this model preformed well with that turned on.

Each hyperparameter instance was trained 6e5 timesteps and evaluated on 10 episodes. The best score reached 318 with parameters:

```
    Trial 20 finished with value: 318.51089287815773 and parameters:
{'learning_rate': 7.613357745707285e-05, 'buffer_size': 952203, 'batch_size':
256, 'tau': 0.014397519335421863, 'gamma': 0.9660841833980763, 'train_freq': 1,
'gradient_steps': 13, 'ent_coef': 0.001}. Best is trial 20 with value:
318.51089287815773.
```

The real model was then trained with these parameters. The score of 300+ was reached in around 430 000 timesteps, which took 1.9 hours, which can be seen in Figure 5.



	Run ↑	Smoothed	Value	Step	Relative
●	sac_walker_v2.zip_0	832,4988	831,7	489 648	2.351 hr

Figure 5. Training of SAC bipedal walker (not overnight). Score of 300+ was reached in around 430 000 timesteps which took 1.9 hours. It nearly hit 300 already around 340k timesteps.

Comparing the two methods, we can see the off-policy method was much more sample efficient. Hitting score of 300 in around 400 000 steps while the on-policy took 3.1 million timesteps. A huge difference.

Another big difference is on the computational costs. Off-policy method took 1.9 hours while the on-policy took just 18 minutes, which is crazy fast.

Both methods earn their respective place in the reinforcement learning catalogue. If gathering data is no problem (like in simulations), on-policy PPO is the way to go, but if the amount of data is limited, off-policy SAC is the choice. Both models learned in a very stable way, although you can see a fair bit more fluctuation with the on-policy SAC model. When evaluating the models, both performed consistently across different seeds.

This assignment was very interesting, and I feel I learned a lot. Thank you for the course!