FACULTY OF ENGINEERING
AND ARCHITECTURE

Bottom-up Exploration of Curriculum Learning for Robotics Control

Thomas Lips
Student number: 01608772

Supervisor: Prof. dr. ir. Francis wyffels
Counsellors: Andreas Verleysen, Victor-Louis De Gusseme

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT
UNIVERSITY

FACULTY OF ENGINEERING
AND ARCHITECTURE

Bottom-up Exploration of Curriculum Learning for Robotics Control

Thomas Lips
Student number: 01608772

Supervisor: Prof. dr. ir. Francis wyffels
Counsellors: Andreas Verleysen, Victor-Louis De Gusseme

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

GHENT
UNIVERSITY

Permission of Use

Acknowledgements

Throughout the writing of this master's thesis, I have learned more than I could have hoped for. I had the opportunity to get acquainted with Deep Reinforcement Learning and its application in robotics control, which I found incredibly exciting. Furthermore, and even more importantly, I learned a great deal about scientific research and methodology, and about myself. None of this would have been possible without the support and assistance that I received, not only throughout this thesis but during my entire studies these past five years. Therefore, some words of gratitude are in place.

To my supervisors Francis, Andreas and Victor-Louis, for introducing me to Industrial Robotics and Deep Reinforcement Learning (its fascinating achievements as well as its more daunting aspects), both of which were completely new to me. For being so involved in the entire process and for providing a stimulating environment. Finally, for having a critical and multi-disciplinary view on science.

To Bruno Huysmans, for helping me discover the fascinating worlds of robotics and programming in the context of the Dwengo robot competition, in which I participated under his supervision many years ago. This experience was the motivation to pursue an engineering degree.

To my friends, for sharing the load, providing perspective and for the countless good memories.

To my parents, for their continued support and guidance. Not only during these past years but throughout my entire life.

Lastly, to Clara, for everything and so much more.

Bottom-up Exploration of Curriculum Learning for Robotics Control

Thomas Lips

Supervisor: Prof. dr. ir. Francis wyffels
Counsellors: Andreas Verleysen, Victor-Louis De Gusseme

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2020-2021

Abstract

Curriculum learning is becoming increasingly popular for Deep Reinforcement Learning to address issues related to exploration, generalization and sample efficiency. In this work, goal curriculum learning for multi-goal robotics problems with continuous action spaces is explored. This is done in a bottom-up fashion, where we describe the entire process of solving a custom robotics problem with Deep Reinforcement Learning and discuss some lessons learned along the way. We focus on generating curricula using asymmetric self-play, where a teacher sets the goals for the student by reaching them from a shared initial environment state. This framework has a number of potential issues, in particular with continuous action spaces where policies learn a unimodal action distribution, resulting in convergence of the teacher to certain regions of the goal space. We first implement and evaluate two state-of-the-art learning algorithms. These are then used to create an implementation of the asymmetric self-play framework. Furthermore, a replay buffer is introduced to the framework to mix the goals proposed by the teacher. The performance of this framework is then explored using a custom motion planning learning environment. From these experiments, we conclude that the convergence leads to overfitting of the student on the goals proposed by the teacher, which results in the system getting stuck. Introducing the replay buffer does not completely solve this problem. Future work for overcoming the observed issues is then discussed. Finally, the framework is also compared to an alternative curriculum method, hindsight experience replay, which is considerably less complex and results in a good performance on the motion planning task.

Keywords — Curriculum Learning, Asymmetric Self-Play, Deep Reinforcement Learning, Robotics

Bottom-up Exploration of Curriculum Learning for Robotics Control

Thomas Lips*

Supervisor: Prof. dr. ir. Francis wyffels
Counsellors: Andreas Verleysen, Victor-Louis De Gusseme

***Abstract* – Curriculum learning is becoming increasingly popular for Reinforcement Learning to address issues related to exploration, generalization and sample efficiency. In this work, we explore the use of goal curriculum learning for multi-goal robotics problems with continuous action spaces. We focus on asymmetric self-play, where a teacher sets the goals for the student by reaching them from a shared initial environment state. This framework has some potential issues, in particular with continuous action spaces where policies learn a unimodal action distribution resulting in convergence of the teacher in the goal space. The impact of these issues is explored using a motion planning learning environment, from which we conclude that the convergence results in overfitting of the student on the current goals proposed by the teacher. We introduce a replay buffer for the student to mix these goals but find that it does not completely solve the issues. Finally, future work for overcoming the observed issues is discussed and the framework is also compared against an alternative curriculum method, hindsight experience replay.**

***Keywords* — Asymmetric Self-Play, Curriculum Learning, Reinforcement Learning, Robotics**

I. INTRODUCTION

Model-free Deep Reinforcement Learning (DeepRL) has been shown a promising learning framework for complex robotics control problems in recent years [1], [2]. However, DeepRL faces several issues that make it challenging to apply to real-world robotics. These issues include delayed rewards, sample inefficiency and generalization to related tasks [3].

To overcome the explorational difficulties caused by delayed rewards, additional reward signals are often added. However, this can easily lead to local optima, which makes learning from the natural binary reward of interest for robotics [4], [5]. Generalization on the other hand is often achieved by randomizing certain aspects of the task, which can also reduce performance and further decreases the efficiency [6], [7]. Curriculum learning has been used increasingly to guide both exploration and generalization for DeepRL [8]. This work focuses on curriculum learning for goals, which is one of the aspects that can be controlled by curricula [8].

Asymmetric self-play (ASP) is a teacher-student framework for creating such goal curricula that has the potential to guide agents for exploration, generalization and even discovery of the goal space. The framework is illustrated in Fig. 1. However, the unimodal policy representation for agents in continuous action spaces has been suggested to lead to convergence issues for ASP [9]. This work evaluates ASP, compares it to another curriculum method and based on the resulting findings, proposes extensions to the framework to overcome the observed issues.

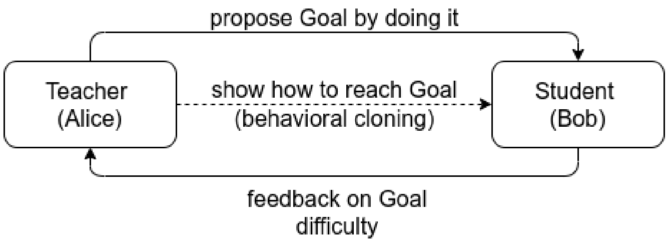

propose Goal by doing it
Teacher (Alice)
show how to reach Goal (behavioral cloning)
Student (Bob)
feedback on Goal difficulty

Figure 1: Schematic overview of the agents and their interactions in the asymmetric self-play framework

II. RELATED WORK

Asymmetric self-play was first introduced in [10] as an addition to regular RL training to improve exploration. The novelty of the approach lies in the way the teacher Alice proposes the goals to the student Bob: she does this by reaching the goal from the shared initial state of the environment. Alice is incentivized to reach goals of an appropriate difficulty by a time-based episode reward: $\gamma \max(0, t_B - t_A)$. The authors interleaved regular play with 10% self-play on a number of tasks with discrete action spaces and delayed dense rewards, where they found that this framework increased the performance by increasing the explorational capacity.

The same idea was more recently used by OpenAI to learn generalized manipulation behavior using only self-play [11]. The authors added behavioral cloning (BC) to the framework, which allows Bob to learn directly from the trajectory Alice used to reach the goal. This was found to be an essential addition. Furthermore, they used binary rewards for both Alice and Bob. Again the framework was used in discrete action spaces with on-policy methods for Alice and Bob.

In other work on goal curriculum learning, Florensa et Al. argue that using ASP in continuous action spaces does not work as the unimodal policy representation would result in Alice converging to small parts of the goal space at each time [9]. This would then result in Alice getting stuck as Bob learns to solve the goals she is proposing. As a consequence, the framework is unstable and results in low performance.

III. BACKGROUND

A. Deep Reinforcement Learning

Reinforcement learning (RL) is a framework for learning optimal strategies for agents in environments that are modelled using a (fully-observable) Markov Decision Process (MDP) $T = \langle S, A, P, r, \rho_0 \rangle$. The agent observes at each timestep a state $s_t \in S$ and has to decide on an action $a_t \in A$. This action then

*thomas.lips@ugent.be

updates the environment state according to $p(s_{t+1}|s_t,a_t)$ and leads to a reward $r(s_t,a_t,s_{t+1})$. Initial states for each episode are sampled according to ρ_0.

The agent's objective is to learn a policy $\pi(a|s)$ that maximizes its discounted expected return, given by

$$J(\pi) = \mathbf{E}_{\tau\sim\pi,s_0\sim\rho_0}\left[\sum_{t=0}^{T}\gamma^t r(s_t,a_t,s_{t+1})\right], \tag{1}$$

where τ represents a trajectory of the agent starting from an initial state: $(s_0,a_0,s_1,a_1,...)$.

In [12] a goal-based extension to the MDP is proposed for multi-goal tasks. To this end a goal space G is added to the MDP from which goals are sampled according to ζ. The reward function now additionally depends on the current goal g. Furthermore as in [4], a mapping $m: S \to G$ from each state to a corresponding goal is assumed.

Finally, in Deep Reinforcement Learning (DeepRL) neural networks are used as function approximators. These networks are then typically optimized using gradient descent.

B. Curriculum Learning

The idea behind curriculum learning is that the training process might benefit from structuring the training data in a certain way, e.g. by starting with easier samples and then moving on to more difficult samples [8], [13].

Curricula can be introduced for various aspects of the task T, but this thesis focuses on goal curricula which vary the goal distribution ζ. Whereas these curricula used to be hand-designed or were based on heuristics, more recently the curricula C themselves are learned concurrently to maximize the agent's performance on the target task distribution T_{target} by optimizing

$$\max_{C} \mathbf{E}_{T\sim T_{target}}\left[J(\pi|T)\right]. \tag{2}$$

IV. METHOD

Our objective is to generate goal curricula using asymmetric self-play. To this end, we create an implementation of the asymmetric self-play framework that is shown in Fig. 1. We then construct a simulation environment for a motion planning task using continuous joint control for a UR3e industrial robot, which we use to explore the framework. More details on the framework, environment and experimental setup are provided in the next subsections.

A. Asymmetric Self-Play

In an attempt to anticipate on the convergence of Alice we used Twin-Delayed Deep Deterministic Policy Gradient (TD3)[1] [14] for Bob. This is an off-policy algorithm, which allows for using a replay buffer to mix the goals proposed by Alice so far. For Alice we used Proximal Policy Optimization (PPO) [15], which is the go-to on-policy algorithm. Bob's policy takes as an input the state and the goal provided by Alice, whereas Alice's policy takes as input the current state as well as the initial state of the environment.

As we use a multi-goal formulation, a goal can be extracted from the final state reached by Alice. This goal can then be presented to Bob, which allows for simply using the environment reward for Bob instead of an internal reward as in [10]. For Alice, we use the time-based reward function proposed in [10] but remove the lower bound in an attempt to avoid Alice getting stuck as reported in [9]. The resulting episode reward for Alice is then given by $0.01(t_B - t_A)$. For Alice to control her episode duration, a STOP action needs to be encoded. Unlike the original framework where this was done with a separate action head, we simply use the action norm: $STOP = \frac{||a||_2}{action\_dim} < 0.2$. Note that this does not allow Alice to take fine-grained actions and might be problematic for other tasks than the motion planning task we consider.

Finally as proposed by OpenAI [11], we also evaluate behavioral cloning (BC) on Alice's trajectories. Since a replay buffer is used, we simply add Alice's trajectory to this buffer if Bob did not manage to reach the goal.

The resulting pseudo-code for asymmetric self-play can be found in Algorithm 1. The entire codebase is available online[2] for more details about the framework or individual learning algorithms.

Algorithm 1 Asymmetric self-play

```
Initialize A: Alice's PPO policy, B: Bob's TD3 policy, E: Environment, $D$: Bob's replay buffer
for N episodes do
    get initial state $s_0 \sim \rho_0$
    $t_A = 0$
    while $a_{t_A} \neq STOP$ and $t_A < t_{MAX}$ do
        $a_{t_A} \sim \pi_A(\cdot|s_{t_A}||s_0)$
        get $s_{t_A+1}$, $r_{t_A}$ from E using $a_{t_A}$
        $t_A = t_A + 1$
    end while
    $g = m(s_{t_A})$
    $t_B = 0$
    while goal not reached and $t_B < t_{MAX}$ do
        $a_{t_B} = \pi_B(s_{t_B}||g) + \varepsilon$
        get $s_{t_B+1}$, $r_{t_B}$ from E using $a_{t_B}$
        $t_B = t_B + 1$
    end while
    $r_A = 0.01(t_B - t_A)$
    if Behavioural Cloning active and Bob did not reach goal then
        $\tau_{BC} \leftarrow$relabel $\tau_A$ with $g$
        add $\tau_{BC}$ to $D$
    end if
    set reward of the transitions in $\tau_A$ to $(0,...,0,r_A)$
    train Alice using $\tau_A$
    add $\tau_B$ to $D$ and train Bob using $D$
end for
```

B. Task Description

The simulated environment can be seen in Fig. 4. We modelled a 6DOF UR3e robot using the Unity Engine[3]. This robot is controlled using joint positional control. The task for Bob is to bring the end-effector to a specific position in the Euclidean space, starting from a random initial joint configuration. Using random initialization was found to be important to increase diversity in the goals proposed by Alice. Bob has to control all joints except for a rotational wrist that does not add to the robot's reachable space for this motion planning task.

[1]For TD3, a small implementation error was made and only discovered afterwards: the target noise for the actions is sampled from a uniform distribution instead of a Gaussian distribution. This is not expected to influence the results too much.

[2]`https://github.ugent.be/tlips/thesis-curriculum-learning`

[3]`https://unity.com/`

The action space is limited to $[-3,3]^5$ degrees with a control frequency of 10Hz. Each action sets a new relative target orientation for the PD joint controllers. The state space is 16 dimensional and consists of the absolute orientation and velocity of all 6 joints as well as the linear position of the grippers and the Euclidean position of the end-effector.

The bounds on the joint space are chosen in such a way that the entire quarter sphere with radius (0.1-0.7) m and origin in the robot base is reachable by the robot. This quarter sphere also makes up the target goal space.

C. *Experimental Methodology*

All hyperparameters are listed in Appendix A. For a number of hyperparameters that were empirically found to be important, a random search with 30 runs is performed for each experiment. Furthermore, each experiment is repeated with 5 different random seeds to ensure robustness. To optimize the networks, Adam [16] is used. We report the mean and standard deviation of the performance metric, which is the average success rate over 20 goals sampled uniformly in spherical coordinates[4] from the target goal space. Videos of the learned behaviors are available[5].

V. EXPERIMENTS

A. *Asymmetric self-play with dense rewards*

For this experiment we used a distance based reward function for Bob:

$$r(s,a,s'|g) = ||x_{s'} - g||_2 \,. \tag{3}$$

We compared random goal selection with asymmetric self-play with and without behavioral cloning. The resulting performance can be seen in Fig. 2. From this comparison, it is first of all clear that the reward function does not suffer from local optima and that random goal selection results in a good performance. Asymmetric self-play on the other hand seems to plateau on a lower success rate. Bob's training success rate on the goals proposed by Alice however (Fig. 3) shows that Bob still manages to solve a good portion of the goals proposed by Alice. Furthermore, the resulting curriculum (Fig. 4) of goals proposed by Alice as well as the dips in the success rate show that Alice is still presenting Bob with new goals he does not yet know how to solve.

However, from the same curriculum, it can be seen that Alice is converging to certain parts of the goal space. This suggests that a possible explanation for the stagnation of the performance would be that as Alice starts to converge, Bob is overfitting on these goals which results in no progress on the target goal space.

B. *Asymmetric self-play with sparse rewards*

These experiments use a sparse, binary reward function:

$$r(s,a,s'|g) = 1_{||x_{s'} - g||_2 < 0.05} \,, \tag{4}$$

which makes exploration much harder.

[4]This results in non-uniform sampling in the Euclidean space which makes some goals slightly more valuable than others, a mistake that was only discovered after performing the experiments. It is however expected not to influence the results too much.

[5]`https://youtube.com/playlist?list=PLeFCx883OreTeSIMPKRYskru_fymswiGf`

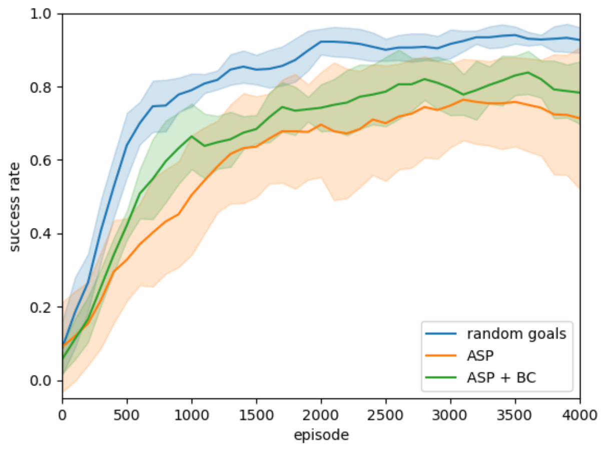


Figure 2: Performance Comparison of asymmetric self-play (ASP), asymmetric self-play with behavioural cloning (ASP + BC) and random goal selection when using dense rewards for the motion planning task.

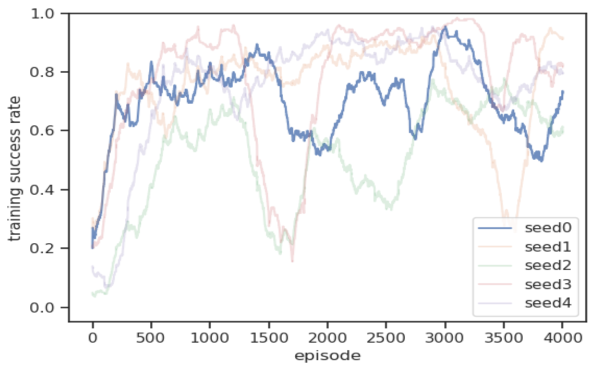
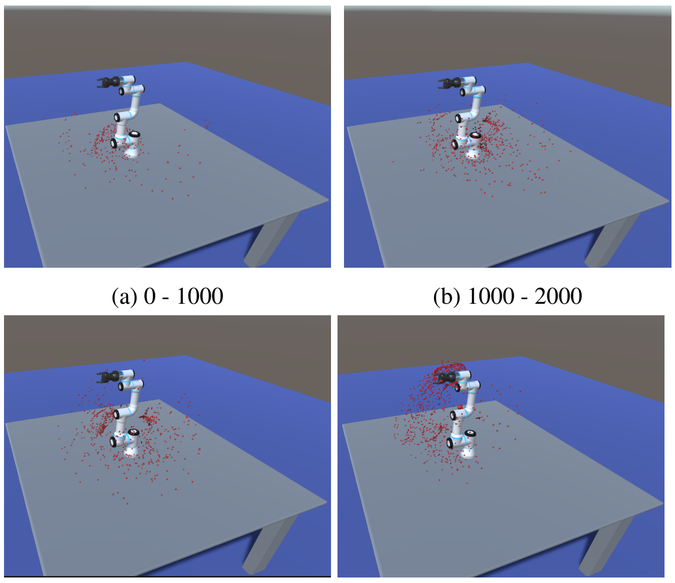


Figure 3: Train success rates for Bob when training with asymmetric self-play (ASP) on dense rewards.



(a) 0 - 1000 (b) 1000 - 2000

(c) 2000 - 3000 (d) 3000 - 4000

Figure 4: Curriculum created by Alice for Bob using dense rewards and asymmetric self-play. Each frame contains the goals proposed by Alice during 1000 consecutive episodes.

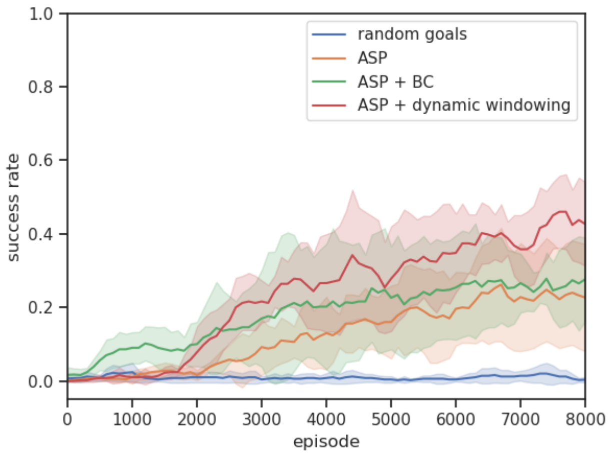


Figure 5: Performance Comparison of random goal selection, asymmetric self-play (ASP), ASP with Behavioural cloning (ASP + BC) and ASP with dynamic windowing when using binary rewards for the motion planning task.

As before we compared random goal selection with asymmetric self-play with and without behavioral cloning. To enforce more diversity and shorter goal distances, we also added a *dynamic windowing* heuristic to limit the maximum episode duration for Alice: whenever Bob reached the previous goal the maximum duration is incremented if Bob did not reach the previous goal it is decremented.

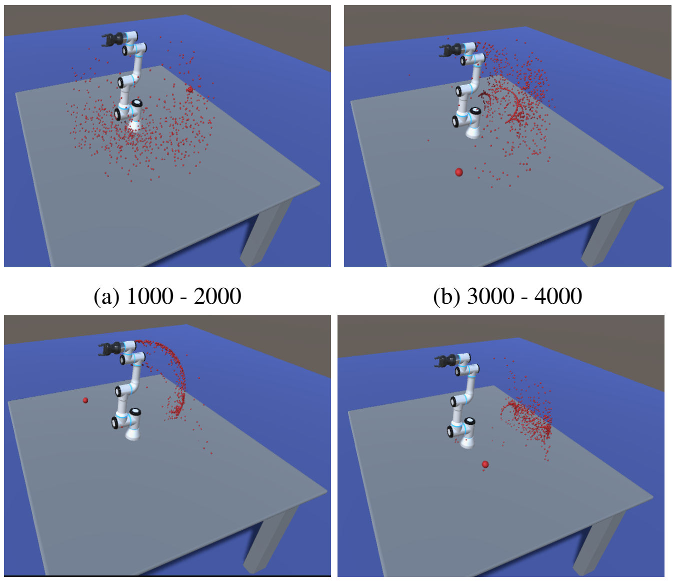
The resulting performance curves can be seen in Fig. 5. As expected random goal selection now results in no learning progress at all, as the probability of reaching a random goal is too small which results in non-informative experiences given the binary rewards.

Asymmetric self-play improves on random goal selection yet plateaus on a low success rate of around 0.2. Adding behavioral cloning does not improve this asymptotic performance although it slightly increases the initial learning speed. Furthermore, the resulting behavior for Bob is often very shaky with behavioral cloning, which is related to the high entropy loss that resulted from the parameter search. This highlights that Alice's trajectories are no 'expert' trajectories, as is usually the case with BC, and should hence be used with care.

Adding the restriction on Alice's episode duration increased the success rate to about 0.4, where it starts to stagnate. From the curriculum proposed by Alice (Fig. 6) it can be seen that she indeed starts to converge later on, which will make Bob overfit on the goals proposed by Alice and most likely explains this stagnation.

C. *Hindsight experience replay with sparse rewards*

At this point, it seemed that the tendency of Alice to converge limits the performance for asymmetric self-play. A method that presents Bob with more diverse goals should give better results. To this end we also evaluated hindsight experience replay (HER) [4]. We used the `future` replay strategy and replayed each transition with 4 goals. For this experiment, the HER + TD3 implementation of stable baselines [17] was used. Hyperparameters were chosen to match the ASP hyperparameters or set to their default values and are included in Appendix A. Unlike previous experiments, the performance is evaluated against the number of steps taken by Bob in the environment. Note that this does not take the additional steps taken by Alice into account for ASP.



(a) 1000 - 2000 (b) 3000 - 4000

(c) 5000 - 6000 (d) 7000 - 8000

Figure 6: Curriculum created by Alice for Bob using binary rewards and asymmetric self-play with dynamic windowing to restrict Alice's episode duration. Each frame contains the goals proposed by Alice during 1000 consecutive episodes.

The results can be seen in Fig. 7 and clearly show that HER reaches a higher success rate and is considerably more sample efficient.

VI. DISCUSSION AND FUTURE WORK

The curricula that were obtained in our experiments confirm that asymmetric self-play indeed has the potential to learn goal curricula for continuous action spaces in an unsupervised way. We also observed how Alice converges to particular parts of the goal space, as suggested in [9]. Unlike the authors of this work, we do not observe Alice getting stuck as a result. This might be due to the removal of the lower bound on her reward function or because of the use of a replay buffer which partially prohibits Bob from overfitting.

However, next to inherently slowing down the learning speed, the convergence seems to cause overfitting on the goals proposed by Alice. Using a replay buffer to mix previous goals did not solve this and we believe this is caused by the buffer itself becoming too unbalanced. To overcome both issues, we now suggest a few options.

First of all to deal with the overfitting of Bob we see two possible approaches:

- Fall back to on-policy methods for Bob and use an explicit replay mechanism to avoid overfitting. This was successfully used by OpenAI [11] although no details were provided. It also remains the question how this will transfer to continuous action spaces.
- Use non-uniform experience replaying, as was introduced in [18] to increase learning speed. In this case, it would be used to overcome the impact of unbalanced buffer content by selecting the most informative transitions. This should be combined with a better integration of the behavioral cloning trajectories by using separate replay buffers combined with additional direct training of the policy using supervised imitation loss, as suggested in [19].

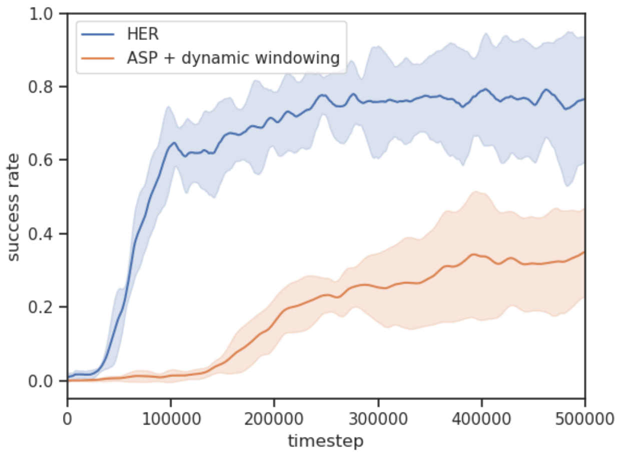


Figure 7: Performance Comparison of asymmetric self-play (ASP) with dynamic windowing and hindsight experience replay (HER) when using binary rewards for the motion planning task.

To limit the convergence (which is the fundamental issue) we suggest two possible extensions:

- Provide Alice with additional reward signals using curiosity-driven exploration [20] to make Alice care more directly about Bob's generalization on the goal space and have her condition the goals more on the initial state.
- Use multiple Alices as was already suggested in [10]. Using Stein Variational Policy Gradients (SVPG) [21], should prevent them from converging to the same regions and increase sample efficiency.

VII. Conclusion

In this work, we explored asymmetric self-play for creating curricula in continuous action spaces. As pointed out in [9], we observed that the unimodal policy of Alice results in convergence of the goals that were proposed. This reduces the learning speed and often even results in complete stagnation as Bob overfits on these goals.

Using a replay buffer with uniform sampling did not solve this, the hypothesis being that the buffer itself becomes too unbalanced by the goals Alice is proposing.

It is expected that with further improvements on the framework these issues can be solved and suggestions hereto were formulated.

The solutions will however make the framework even more complex and harder to analyze. Hence from a pragmatic point of view reward shaping or more heuristic goal curriculum methods such as hindsight experience replay should be tried before turning towards asymmetric self-play.

References

[1] S. Levine, P. Pastor, A. Krizhevsky, *et al.*, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research (IJRR)*, vol. 37, no. 4-5, pp. 421–436, 2018.

[2] I. Akkaya, M. Andrychowicz, M. Chociej, *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[3] J. Ibarz, J. Tan, C. Finn, *et al.*, "How to train your robot with deep reinforcement learning: Lessons we have learned," *The International Journal of Robotics Research (IJRR)*, vol. 40, no. 4-5, pp. 698–721, 2021.

[4] M. Andrychowicz, F. Wolski, A. Ray, *et al.*, "Hindsight experience replay," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017.

[5] I. Popov, N. Heess, T. Lillicrap, *et al.*, "Data-efficient deep reinforcement learning for dexterous manipulation," *arXiv preprint arXiv:1704.03073*, 2017.

[6] B. Mehta, M. Diaz, F. Golemo, *et al.*, "Active domain randomization," in *Conference on Robot Learning (CoRL)*, PMLR, 2020, pp. 1162–1176.

[7] V. Kumar, D. Hoeller, B. Sundaralingam, *et al.*, "Joint space control via deep reinforcement learning," *arXiv preprint arXiv:2011.06332*, 2020.

[8] R. Portelas, C. Colas, L. Weng, *et al.*, "Automatic curriculum learning for deep rl: A short survey," *arXiv preprint arXiv:2003.04664*, 2020.

[9] C. Florensa, D. Held, M. Wulfmeier, *et al.*, "Reverse curriculum generation for reinforcement learning," in *Conference on robot learning (CoRL)*, PMLR, 2017, pp. 482–495.

[10] S. Sukhbaatar, Z. Lin, I. Kostrikov, *et al.*, "Intrinsic motivation and automatic curricula via asymmetric self-play," in *6th International Conference on Learning Representations (ICLR)*, 2018. [Online]. Available: `https://openreview.net/forum?id=SkT5Yg-RZ`.

[11] O. OpenAI, M. Plappert, R. Sampedro, *et al.*, "Asymmetric self-play for automatic goal discovery in robotic manipulation," *arXiv preprint arXiv:2101.04882*, 2021.

[12] T. Schaul, D. Horgan, K. Gregor, *et al.*, "Universal value function approximators," in *International conference on machine learning (ICML)*, PMLR, 2015, pp. 1312–1320.

[13] Y. Bengio, J. Louradour, R. Collobert, *et al.*, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning (ICML)*, 2009, pp. 41–48.

[14] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning (ICML)*, PMLR, 2018, pp. 1587–1596.

[15] J. Schulman, F. Wolski, P. Dhariwal, *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Representation Learning (ICLR)*, 2015.

[17] A. Raffin, A. Hill, M. Ernestus, *et al.*, *Stable baselines3*, `https://github.com/DLR-RM/stable-baselines3`, 2019.

[18] T. Schaul, J. Quan, I. Antonoglou, *et al.*, "Prioritized experience replay," in *International Conference on Representation Learning (ICLR)*, 2016.

[19] A. Nair, B. McGrew, M. Andrychowicz, *et al.*, "Overcoming exploration in reinforcement learning with demonstrations," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 6292–6299.

[20] D. Pathak, P. Agrawal, A. A. Efros, *et al.*, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning (ICML)*, PMLR, 2017, pp. 2778–2787.

[21] Y. Liu, P. Ramachandran, Q. Liu, *et al.*, "Stein variational policy gradient," in *33rd Conference on Uncertainty in Artificial Intelligence*, UAI, 2017.

A HYPERPARAMETERS

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
learning rate	0.001	actor learning rate	[5e-5,1e-3]
ppo clip value	0.2	noise std deviation	[0.1-0.4]
gradient clip norm	0.5	critic learning rate	0.001
batch size	previous episode duration	batch size	[32,64,128]
epochs	10	train frequency	[1-100]/episode
rollout length	previous episode duration	buffer size	1e6
discount factor	0.99	discount factor	0.99
GAE factor	0.95	policy update frequency	2
value loss coefficient	0.5	target noise parameter	0.2
entropy loss coefficient	[0.0-0.001]	target noise clip value	0.5
hidden layer size	128	hidden layer size	[128,256,512]
max episode duration	[100,150,200]	max episode duration	[150,200]

Table 1: Hyperparameters for Alice and Bob on the motion planning environment. For hyperparameters that are included in the hyperparameter search, the range of possible values is given.

	Value						
	Dense Rewards			Sparse Rewards			
Hyperparameter	Random	ASP	ASP+BC	Random	ASP	ASP + BC	ASP + DynWin
Alice							
entropy loss coefficient		0.000735	0.000284		0.000827	0.00358*	0.00108*
max episode duration		100	150		150	100	150
Bob							
actor learning rate	0.000870	0.000897	0.000509	0.000740	0.000667	0.000745	0.000922
noise std deviation	0.380	0.389	0.380	0.280	0.350	0.380	0.385
batch size	128	128	128	32	64	64	128
train frequency /episode	55	91	60	76	75	95	35
hidden layer size	512	512	512	512	128	128	256
max episode duration	200	200	150	250	150	200	150

Table 2: Hyperparameters obtained with the random search. Note that for some experiments the range for the entropy loss coefficient was by mistake set to $[0-0.005]$ instead of $[0-0.001]$. This is indicated with an asterisk.

Hyperparameter	**Value**
learning algorithm	TD3
learning rates	0.001
noise std deviation	0.4
train frequency	50 / episode
batch size	64
replay strategy	*future*
replay goals / transition	4

Table 3: Hyperparameters for Hindsight Experience Replay.

Table of Contents

List of Figures

List of Tables

Nomenclature

Acronyms / Abbreviations

ACL	Automatic Curriculum Learning
ASP	Asymmetric Self-Play
BC	Behavioral Cloning
DDPG	Deep Deterministic Policy Gradient
DeepRL	Deep Reinforcement Learning
GAE	Generalized Advantage Estimation
HER	Hindsight Experience Replay
MDP	Markov Decision Process
PER	Prioritized Experience Replay
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SVPG	Stein Variational Policy Gradient
TD3	Twin-Delayed Deep Deterministic Policy Gradient
TD	Temporal Difference

Chapter 1

Introduction

This master's thesis aims to explore the use of asymmetric self-play [1] for curriculum learning in continuous robotics control with model-free Deep Reinforcement Learning. Asymmetric self-play is a teacher-student framework in which the teacher tries to aid the student in learning to solve a particular multi-goal problem. This is done by creating a useful order in the goals presented to the student during training, which is referred to as goal curriculum learning. Curriculum learning has become very popular recently as a way to guide robots during exploration or to guide the randomization that is often used in an attempt to increase the generalization of learned policies.

In the next sections the context, which was briefly summarized above, will be elaborated upon, before stating the research objectives and giving an overview of the different chapters of this work.

1.1 Problem Statement

Industrial Robots[1], which are also referred to as *robotic arms* or *robotic manipulators*, are used increasingly across many domains of the industry to perform dangerous or difficult jobs and to increase efficiency [2]. The behavior of these robots is usually meticulously hand-engineered upfront using analytical methods such as inverse kinematics and control theory. This approach is time-consuming, requires expert domain-knowledge and limits the range of tasks for which robots can be used. Programming Industrial Robots for example to unpack items in warehouses is not feasible due to a.o. the high variability of these items (softness, size...) and their packaging (materials, size, shape..).

Recently, instead of programming the behavior, researchers have started to work on developing systems that can learn the desired behavior by interacting with their environment, resulting in more domain-agnostic control solutions. Deep Reinforcement Learning (DeepRL) is an approach to this problem that combines Reinforcement Learning with neural networks. The use of these powerful function approximators has made it possible to scale the learning to high-dimensional

[1]Whenever the word robot is used in this thesis, it refers to these Industrial Robots

problems [3]. Model-free DeepRL is a completely data-driven direction within DeepRL that has shown great potential for robotics control in recent years [4, 5].

However, model-free DeepRL still faces some issues that limit its application to many real-world robotics problems [6]. Three of these issues, that are particularly relevant for this thesis, are delayed rewards, sample inefficiency and generalization to related situations. These issues are briefly introduced in the next paragraphs.

Learning suffers from delayed rewards, which characterize most robotics problems. A policy might have to take quite a number of steps before solving a task and receiving the accompanying reward. This makes exploration an important issue. One approach to deal with this problem is reward shaping, in which the "natural" reward signal, which is usually a binary indicator of whether the robot achieved its task, is replaced or combined with a more informative reward signal. This however can easily introduce local optima and hence reduce the performance of the learned behavior. Either way, the optimization landscapes in DeepRL tend to be rather hard [6].

Next to being hard to train, sample efficiency of model-free learning methods is in general quite poor, often requiring millions of interactions with the environment before learning to solve a problem. Complicating this even more is the aforementioned need for exploration, which makes real-world data collection not always safe for the robot or its environment. To mitigate these problems, researchers often make use of simulation environments to train policies. Using simulators, however, inevitably creates a *reality gap*, making transferring the learned policies to real-world robotics challenging. Overcoming this *reality gap* requires policies that can generalize, or deal with problems related but different to those they were trained on.

Generalization can require the policy to deal with changes in the environment (e.g. the obstacles the robot can face, their location, etc.), the goal (e.g. location to reach with the robot arm or target position for the item it has to pick-and-place), or properties of the actuators and sensors (e.g. the dynamics of the robot and its environment or the perception characteristics). This last category in particular is important for closing the *reality gap* and making Sim2Real transfers possible. Generalization however is yet another issue for DeepRL policies, which tend to be very "narrow-minded" in the sense that they do not perform well on unseen variations on the aforementioned domains. As a result, these policies often have very limited applicability in the real world, where variations in the environment are often inherent to the task, such as for the use case of unpacking items that was introduced before. To deal with this issue, the training data is typically randomized over the desired domains in the hope of making the policy induce the desired, generalized behavior [7]. Too much randomization however has been found to lead to decreased performance and stability issues [8, 9], next to decreasing sample efficiency even more.

To tackle these issues, curricula have been used increasingly to guide the randomization or exploration by presenting the policy at each moment of its training with a set of appropriate environments, goals and/or physical properties. Such curricula used to be engineered upfront but more and more this is replaced by frameworks for automatically learning the curricula [10].

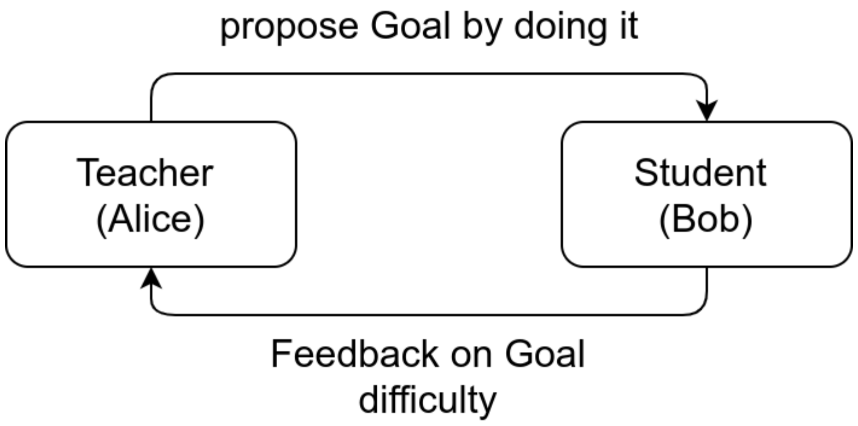


Fig. 1.1 High-level illustration of the Asymmetric Self-play framework.

This thesis focuses on curriculum learning for goal selection. Curriculum learning in this context can be used for three (often overlapping) reasons [10]:

1. Organising exploration for solving task(s) that cannot be solved directly because they are too hard or the reward is too delayed, which will be referred to as curriculum learning for exploration.
2. Training agents to generalize to a known goal space, or multi-goal curriculum learning.
3. Organising open-ended exploration of the goal-space if it is not known or cannot be described easily (which is required to sample from it). This will be referred to as curriculum learning for goal-discovery.

Asymmetric self-play [1], is an elegant approach for generating such goal curricula. In this method, a teacher, referred to as Alice, is introduced who proposes goals to the student, Bob, by reaching them from the shared initial state of the environment. The student Bob then provides feedback on the difficulty of this goal to the teacher and this enables the teacher to create a curriculum of appropriate goals. These interactions are illustrated in Fig. 1.1.

This framework was initially introduced to increase the explorational capacities of the student Bob. More recently however, OpenAI built on top of this idea and used it for goal-discovery, in which a policy for a range of manipulation tasks was learned unsupervised, using self-play [11]. This shows that Asymmetric self-play has the potential to combine all three reasons for using goal-curricula within a single framework, which is highly attractive.

However, all previous work has been done on discrete action spaces. This implies that Alice was able to learn a multimodal distribution over the action space. As suggested by Florensa et al. [12], using continuous action spaces for Alice might pose additional difficulties as this would imply Alice has to learn a unimodal distribution over the action space, since continuous policies are always parameterized with a Gaussian distribution. This would limit her capabilities to propose a wide range of goals to Bob as she would only be able to move to a particular subset of the goal space at a time.

Continuous action spaces often come naturally in robotics problems, since the real world is continuous by nature. One can often discretize the action space, but this approach suffers from the curse of dimensionality, leading to a rapidly increasing number of actions if the dimension of the action space becomes larger even with the coarsest of quantizations [13]. This makes continuous action space handling of great interest for robotics control.

1.2 Research Objectives

The main objective of this thesis is to explore the idea of asymmetric self-play for creating goal curricula in the context of robotics control with continuous action spaces.

Asymmetric self-play is an elegant method that could potentially tackle all the three motivations for using goal curricula but comes with the explicit caveat that combining it with continuous action spaces might create additional issues. In this thesis, the impact of these issues will be evaluated. Furthermore, some modifications and extensions to the framework to reduce their influence will be formulated and explored.

A secondary objective is to explicitly explore this idea in a bottom-up fashion. The motivation is that such a bottom-up approach will provide more insights into the asymmetric self-play framework on the one hand but also provide useful information on the process of using DeepRL frameworks to attempt to solve custom problems.

1.3 Outline

This work contains six chapters (not taking the introduction and conclusion into account), that build towards evaluating the asymmetric self-play framework for controlling a custom robot setup with continuous action spaces. This will be learned in a simulated environment to avoid issues with real-world data collection and efficiency.

In Chapter 2, the necessary background on DeepRL is provided together with a short overview of related work on asymmetric self-play and related methods for goal curriculum learning.

Details about the simulation environment that was created with the unpacking use case that was introduced before in mind, are provided in Chapter 3.

The different learning tasks that are used are described in Chapter 4, together with other elements of the experimental setup.

Chapter 5 introduces the learning algorithms that will be used later on for the teacher, Alice, and the student, Bob, in the asymmetric self-play framework. This chapter also focuses on the implementation and testing of these algorithms.

In Chapter 6, the self-play framework is described in more detail and its potential issues are discussed together with some adaptations that might mitigate them. Then, using the learning algorithms that were implemented in the previous chapter, an implementation of this framework is described. Finally, the framework is applied to a low-dimensional task to evaluate the occurrence of these issues and the effectiveness of the proposed adaptations.

After gaining some intuition, exploring the limitations and some possible strategies for mitigating them, the framework is used in Chapter 7 to learn a motion planning task using the constructed simulation environment. From this chapter, it will become clear that the formulation of asymmetric self-play as constructed in Chapter 6 is still hindered by some of the described issues and is hence limited in its performance. Solutions for these issues are discussed and the framework is also compared with hindsight experience replay, another method for generating curricula.

Finally, since this thesis also aims to provide insight into the process of trying to solve a custom problem with a DeepRL framework, some chapters contain a section with a number of *Lesssons Learned*. Some of these are in hindsight rather obvious and most of them will not come as a surprise to researchers with experience in DeepRL. However as this research discipline is still quite new and evolving very fast, information tends to be very fragmented and quite some knowledge or best practices only live in the minds of the researchers working on it, which makes these lessons learned hopefully still informative.

Chapter 2

Background and Related Work

This chapter briefly introduces the necessary background on Reinforcement Learning and some related work on goal curriculum learning. The first section introduces the mathematical background and notation for Reinforcement Learning, as used in this thesis. The second section introduces curriculum learning and the third section provides a short overview of the work on asymmetric self-play and some related methods for the generation of goal curricula.

2.1 Reinforcement Learning

In this section, the Reinforcement Learning (RL) framework is briefly introduced. First, the necessary notation is introduced. Then an extension to the RL framework is given and the use of neural networks as function approximators is briefly introduced. Finally, a short overview of some characteristics of different model-free learning methods is provided.

2.1.1 Formulation

Reinforcement Learning deals with agents that interact with their environment and have to maximize the expected reward of their actions. This reward defines the task the agent has to solve [14]. The process of optimal decision making is formalized as a Markov Decision Process (MDP) and referred to as a task T:

$$T = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho_0 \rangle \,. \tag{2.1}$$

Here $\mathcal{A}$ represents the action space, which can be discrete or continuous, and from which the agent can select its action at each step of the process. The state space $\mathcal{S}$ describes the entire state of the environment, in such a way that it obeys the Markov property. $\mathcal{P}(s_{t+1}|s_t, a_t)$ is the transition function that describes the influence of the agent's actions on the environment state given the Markov property for the state. $r(s_t, a_t, s_{t+1})$ is the reward function and ρ_0 the distribution of the initial state s_0 of the environment. To reduce the notational overload, the current state is often referred to as s instead of s_t whereas s' is a common notation for s_{t+1}. The same goes for the actions and rewards.

A trajectory consists of a sequence of actions and the resulting states, starting from an initial state $s_0 \sim \rho_0$:

$$\tau = (s_0, a_0, s_1, a_1, ..) \,. \tag{2.2}$$

A single step from a trajectory is often referred to as an experience or a transition and is given by the tuple (s, a, r, s', d) where d indicates if the transition was terminal.

The agent's objective is to maximize the expected return $R(\tau)$, which is given by:

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t \,, \tag{2.3}$$

where γ is an optional discounting factor and T is the time horizon, which could be infinite.

To do so, the agent has to learn a policy $\pi(a|s)$ by maximizing

$$J(\pi) = \mathbb{E}_{\tau \sim \pi, s_0 \sim \rho_0} \left[R(\tau) \right] \,, \tag{2.4}$$

where $\tau \sim \pi$ indicates that the trajectory is obtained by following the policy π starting from the initial state s_0. The optimal policy is then given by $\pi^* = \mathrm{argmax}_\pi J(\pi)$.

There are two other functions that are often learned next to the policy: The value function $V(s)$, which is given by

$$V(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s] \,, \tag{2.5}$$

and the Action-Value function (or Q-function) $Q(s, a)$, which is defined as:

$$Q(s, a) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \,. \tag{2.6}$$

The section is concluded with two simplifications of the RL framework as it was introduced here:

Fully Observable Markov Decision Process

The formal description that has been introduced in this section implicitly assumed a fully observable MDP. This implies that the agent's observation o_t of the environment provides him with information of the full state s_t of the environment. In this case, to easy notations, the observation is usually left out of the formulation and the agent's input is denoted as s_t as was done in this section and throughout this thesis.

Model-free RL

In this thesis, only model-free RL will be considered. In model-free RL methods, one learns directly the policy and/or value functions. This is in contrast with model-based RL, where usually the transition and/or reward function is learned and then used with other planning techniques to maximize the return.

2.1.2 Multi-Goal Formulation

An extension to the RL framework was provided by Schaul et al. [15] by introducing a multi-goal formulation of the MDP process. In this case the reward function is conditioned on a goal $g \in \mathcal{G}$: $r(s, a, s'|g)$. The resulting MDP is then given by:

$$T = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho_0, \mathcal{G}, \zeta \rangle \,, \tag{2.7}$$

where $\zeta(g)$[1] is a distribution over the goal space.

The objective is still to learn a policy $\pi(a|s, g)$ that maximizes

$$J(\pi) = \mathbb{E}_{\tau \sim \pi, s_0 \sim \rho_0, g \sim \zeta} \left[R(\tau) \right] \,. \tag{2.8}$$

As in [16], a mapping $m : \mathcal{S} \to \mathcal{G}$ from each state to a corresponding goal is assumed in this work. In the most trivial case, this is simply an identity function.

2.1.3 Deep Reinforcement Learning

Deep Reinforcement Learning (DeepRL) tries to solve decision problems under the RL framework as introduced before by using neural networks as function approximators for learning optimal policies, value functions, etc. It was successfully used for the first time on high-dimensional problems by Mnih et al. [17] and has since attracted much interest. To indicate the use of neural networks as function approximators the functions usually get a subscript that refers to the parameters of the network:

$$\pi_\theta(a|s) \approx \pi^*(a|s) \,. \tag{2.9}$$

These parameters are usually optimized using gradient-based optimization, although evolutionary strategies are also possible. In this work, the gradient-based Adam optimization algorithm [18] is used.

2.1.4 Short Taxonomy of Model-Free Deep Reinforcement Learning Algorithms

There exist many different model-free DeepRL learning algorithms, which can differ greatly in many aspects. Four properties that can be used to classify them, are briefly introduced below.

A first classification is based on what is primarily learned. This divides the algorithms into two groups: Policy Gradient methods which directly learn a policy π and Q-learning methods which learn a Q-function $Q(s, a)$. For various reasons, often both a value function or Q-function and a policy function are learned at the same time and this is usually referred to as *Actor-Critic* algorithms. However, there is always one of the two that is the main focus of the learning algorithm.

[1]This notation was chosen arbitrarily as it is often implicitly assumed that goals are sampled randomly and hence no distribution is specified. OpenAI uses $\mathcal{G}(g)$ [11] but this seemed confusing as $\mathcal{G}$ denotes the goal space.

Another classification can be made based on the actions that the policy outputs, which can be discrete or continuous. Discrete actions are always represented with a categorical distribution whereas continuous action spaces are represented using a diagonal multivariate Gaussian distribution. In this work, continuous action spaces are always used and hence the policies will always represent such a Gaussian distribution.

The policies can also be either stochastic or deterministic. In the stochastic case, for continuous policies, both the mean and standard deviation are learned. Deterministic policies only represent the mean of the action distribution, implying the need for additional noise to enhance exploration. For deterministic policies, the policy function is given by $\pi(s)$ instead of $\pi(a|s)$ and is also often referred to as $\mu(s)$ to emphasize that this policy is deterministic and encodes the mode.

A final difference lies in whether the samples that are used for updating the learned functions are collected on-policy or off-policy. On-policy algorithms require the samples to be collected by the current version of the learned policy whereas off-policy algorithms can use samples collected using any arbitrary policy, including past versions of the learned policy.

2.2 Automatic Curriculum Learning

Curriculum learning for machine learning was proposed by Selfridge et al. [19] and further popularized by a.o. the work of Bengio et al. [20]. The core concept is that the training process might benefit from structuring the training data in a certain way, e.g. by starting with easier samples and then moving on to more difficult samples. This is based on transfer learning [21], where the idea is that previous training can be reused to increase asymptotic performance or decrease training time. Because of this, curriculum learning is of particular interest for DeepRL, for which some of the core issues are dealing with delayed rewards, improving generalization towards related tasks and increasing sample efficiency [6, 14].

Curricula can be introduced for various aspects of the task T, such as the initial state distribution, goal distribution, transition function, the reward function or even the experiences to replay during training (for off-policy methods) [10]. Whereas these curricula used to be hand-designed or were based on heuristics, more recently the curricula $\mathcal{C}$ themselves are learned concurrently to maximize the agent's performance on the target task distribution T_{target}:

$$\max_{\mathcal{C}} \mathbb{E}_{T\sim T_{\text{target}}} \left[J(\pi|T)\right] \ . \tag{2.10}$$

In this thesis, the focus lies on generating goal-curricula by varying the goal-distribution ζ. There are three use cases for doing so [10]:

1. To improve exploration for difficult tasks or delayed rewards.
2. To increase generalization to a particular goal space $\mathcal{G}$.
3. To organize exploration in case the target goal space is simply not known or can at least not be described mathematically.

2.3 Related Work on Goal Curriculum Learning

This section provides a short overview of related work on goal curriculum learning, with a focus on asymmetric self-play. Many other methods exist and the ones discussed here were chosen because they are most relevant for this thesis. A more general overview can be found in [10].

Hindsight Experience Replay (Andrychowicz et al., 2017)

In this work, goal distributions are implicitly created by a careful selection of the goals against which to replay trajectories during the training phase of off-policy learning algorithms [16]. The motivation is to learn efficiently from binary rewards and avoid the need for domain-specific expert knowledge, which is often required to shape rewards without introducing local optima to the optimization landscape. The problem with binary rewards is that an agent does not receive informative feedback if it did not achieve the goal, as the reward is then the same for all experiences in the trajectory.

The idea is that by introducing the multi-goal RL formulation and defining a mapping between states and goals as was done in Section 2.1.2, trajectories in the replay buffer could be made more informative by replaying them with a different goal. For example, the goal that was achieved in the final state of the trajectory. This way, although the agent might not have been able to reach the intended goal, the trajectory can still be made informative to learn other goals.

The authors showed that even in the case where only one goal is of interest, the performance and learning speed of the agent increased by introducing the multi-goal formulation in combination with hindsight experience replay. Furthermore, it was also shown that badly shaped reward functions can easily hinder the exploration of the agent or create discrepancies between the actual task of the agent and the optimization landscape, further highlighting the importance of learning from sparse rewards to avoid the time-consuming reward shaping process.

Different heuristics for selecting the goals against which to replay each trajectory were discussed and the strategy referred to as `future` was found to perform best across a range of robotics tasks. With this heuristic, k goals that were achieved after the current experience in the trajectory are used to replay this experience with.

Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play (Sukhbaatar et al., 2018)

This paper introduced the idea of asymmetric self-play, in which a teacher (called Alice) proposed goals to a student (called Bob) by reaching them from a shared initial state of the environment (cf Fig. 1.1) [1]. In this work, asymmetric self-play was only used as an addition to speed up exploration and was interleaved with regular training for Bob (90% of the time Bob was using regular play, only 10% of the time Alice proposed goals). No goal-based formulation is used and hence the goals that Alice proposed to Bob are simply encoded as the final state Alice managed to achieve, which means that Bob's policy is conditioned on both the environment state and the state reached by Alice: $a_B \sim \pi_B(\cdot|s_t, s_A)$.

The authors proposed the following reward function for Alice: $r_A = \gamma \max(0, t_B - t_A)$, which is supposed to incentivize Alice to find the easiest goal for her that is still hard for Bob, resulting in goals of appropriate difficulty for Bob. Bob on the other hand receives an internal reward of $-\gamma t_B$, where γ is used to balance the self-play rewards with the external reward function of the environment. In order for Alice to stop her episode (and hence decide which state to present to Bob), an additional STOP action was added to her action space.

This formulation results in a completely unsupervised curriculum, where neither goal space nor any other aspect needs to be described and hence this method could be used for all three use cases of goal curriculum learning.

Although the authors applied the self-play to both discrete and continuous state spaces, they always discretized the action space (even for tasks that are defined with a continuous action space). This has an important implication for Alice, as pointed out in [12] (more details in the next subsection).

Finally, the authors briefly mention that Alice sometimes seems to focus on particular parts of the goal space and propose to use multiple instances of Alice to mitigate this.

Reverse Curriculum Learning for Reinforcement Learning (Florensa et al., 2018)

Florensa et al. introduce another method for creating curricula for exploration [12]. Instead of creating a goal distribution ζ, they modify the initial state distribution ρ (hence the reverse curriculum). The idea here is that the curriculum of initial states starts from the goal state and gradually moves to more distant initial states as the agent's success rate starts to increase.

This curriculum of initial states ρ_i is not learned but evolves by performing random walks in the action space from the current set of initial states. This leverages the idea that the distance between states should be expressed by the number of steps that need to be taken in the MDP to reach the state, referred to as the MDP-distance.

This method comes with a number of downsides: first of all, it can only be used within a single MDP. This is because changing the goal, transition function, or another aspect of the MPD would destroy the relationship between these initial states, which makes this method only suited for the exploration of hard tasks in a single-goal context without any kind of randomization.

Furthermore, this method scales badly in the distance from the desired initial states to the goal state or in the dimensionality of the state space. A proposed solution hereto by the authors is to bias the curriculum towards the desired initial states, similar to the A* extension to the Dijkstra Algorithm.

Florensa et al. also explore asymmetric self-play as an alternative to their method and formulate critique on this framework:

- From some experiments they found that the reward function $r_A = \gamma \max(0, t_B - t_A)$ often leads to uninformative rewards for Alice as Bob might learn faster than she does and hence $t_B - t_A$ becomes negative. They illustrate this with a simple experiment. As stated in [1], this experiment is however not valid as Bob is modeled by a simple heuristic that evolves with non-continuous steps. This does however not mean that the issue they describe could not occur.

- Furthermore they also describe how in continuous action spaces, Alice would be limited to moving into a single direction due to the unimodal Gaussian distribution that is always used to represent continuous action policies. This seems indeed a very legit concern and might explain why the authors in [1] always used discrete action spaces.

Generating Automatic Curricula via Self-Supervised Active Domain Randomization (Raparthy et al., 2020)

In this work, the authors combine an automatic domain randomization method [22] with asymmetric self-play to generate an unsupervised joint curriculum for goals and environments [22]. This seemed interesting for combining the advantages of goal curricula with sim2real generalization to overcome the *reality gap*. The initial research objective of this thesis was hence to use this framework for learning behaviors related to the unpacking of objects, the use case that was briefly introduced before.

However, while exploring their work further, a few issues were found:

- The authors use a rather strange variant of asymmetric self-play in which Bob is actually a time-delayed copy of Alice and where Alice's goal is still simply to reach goals according to ζ. An additional 'stopping policy' is then introduced that decides on where Alice stops and hence what goal she presents to Bob. This approach however seems to be more related to Hindsight Experience Replay in the case where only the final state is used for replay and where an additional network is created for deciding when to stop the current episode. This is no issue per se but seems to introduce unnecessary complexity by framing it as asymmetric self-play.
- Another issue was the apparent presence of a coding bug[2] in the repository that accompanied the paper. This bug would seem to make Alice only propose one-step goals due to an indentation error in a `while` loop[3].
- At the same time, the environment curriculum method results could not be reproduced by others[4].

These observations motivated a switch in the research by taking a step back and focusing only on goal-curricula, in particular on asymmetric self-play.

Asymmetric Self-Play for Automatic Goal Discovery in Robotic Manipulation (OpenAI, 2021)

Attracted by the unsupervised curriculum generation, OpenAI extended the asymmetric self-play formulation recently and used it for automatic goal discovery in robotics manipulation [11]. They

[2]https://github.com/montrealrobotics/unsupervised-adr/issues/3

[3]To make sure I was not missing something I asked the authors for feedback on this, however, I did not receive a response.

[4]https://github.com/montrealrobotics/active-domainrand/issues/6

trained a robot manipulator to bring objects into the desired configuration, which results in a goal space that is hard to describe mathematically and contains goals of varying difficulty. By using asymmetric self-play, the goal space was discovered in an unsupervised way, resulting in policies that performed well across many unseen generalizations using only self-play for training.

The authors started from the formulation in [1] but added Behavioral Cloning to Bob, using the trajectories provided by Alice as experiences for Bob in case he did not manage to reach the goal by himself. This addition was found to be crucial in an ablation experiment. Furthermore, the original reward function for Alice was replaced by a simple binary reward, as this was found not to influence the resulting performance. As in [1], they use discrete action spaces for the control policies of Alice and Bob. A final addition is that they make Bob and Alice also train against past versions of their opponents.

Conclusion
It is clear from this overview that asymmetric self-play is a very appealing method for creating goal curricula. It can be used for all three use cases for goal-curricula. It does not generate infeasible goals as all goals are proposed by doing them, does not require the goal space to be described mathematically and uses by construction an appropriate distance measure: the MDP-distance. However, some researchers found that the unimodal policy representation of continuous action spaces increases the tendency of the system to get stuck as the teacher Alice converges, although without providing a thorough analysis or much details. The goal of this thesis is hence to explore the issues with asymmetric self-play in continuous action spaces, see how they limit performance and start on formulating solutions to overcome them.

Chapter 3

Robot Simulation Environment

This chapter describes the Simulation Environment that was created to learn behaviors with a specific robotics use case in mind: the unpacking of items using a Universal Robotics UR3e[1] industrial robot. As this use case comes with some specific needs that are not covered by existing environments for robotics learning such as the often used OpenAI Robotics environments [23], a custom simulation environment was built using the Unity Engine[2]. This simulation environment allows to create different learning tasks and environments for the UR3e.

The Simulation Environment consists of three high-level components, as shown in Fig. 3.1. These components are the simulation world, the ML-Agents component and the Python Environment Interface. Each of these components is discussed in more detail in sections 2-5. First, some design choices for the environment are discussed. The chapter concludes with some lessons learned during the implementation of the simulation environment.

[1]https://www.universal-robots.com/products/ur3-robot/
[2]https://unity.com/

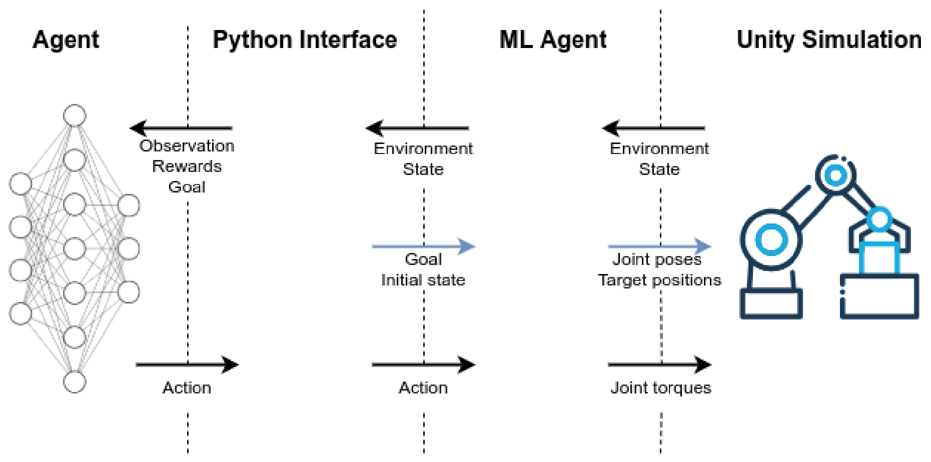


Fig. 3.1 High-level components and their interactions. The agent itself is also included to provide an overview of all interactions. Blue arrows indicate sidechannel communications.

3.1 Environment Design Choices

Before the technical details of the environment are elaborated upon, the design choices for the environment are discussed.

Simulator

Unity was chosen over existing alternatives such as MuJoCo [24] and Pybullet [25], which are used more often in the robotics community. Unity was created as a game engine at first and hence focused more on rendering than on physics. Recently, however, Unity Engine has started to focus more on robotics. They recently integrated the PhysX4.0 physics engine[3], which uses reduced coordinate systems for simulating articulation chains. This greatly improves the accuracy of robotics simulation.

Combining this with their ML-Agents framework [26] for facilitating reinforcement learning of simulated agents, the large and active community, the high-end rendering capabilities and the open-sourced codebase it has become an interesting alternative for the more expensive MuJoCo.

Control

Controlling a robot can be done in more than a few ways depending on both the abstraction and precision level of the actions that the policy has to pass to the robot.

Regarding the abstraction level, one has broadly three options:

1. Joint Torque Control,
2. Joint Positional Control,
3. End-effector Positional Control.

The UR3e is a high-end and standardized industrial robot. This makes end-effector control indeed a very interesting choice as it allows to leverage prior knowledge about the (inverse) kinematics to control the end effector directly. However, this also has the downside that one can no longer make full use of the overarticulation of the Robot (the UR3e has 6 degrees of freedom), which limits the capacity of the robot to for example reach into a box or paper bag.

With joint torque control, the policy actions determine the torques on all joints. This low-level control strategy does not leverage the high quality and accuracy of the UR3e hardware. The UR3e comes with built-in controllers that can reproduce[4] trajectories with an accuracy of about 0.03 mm, which requires high-precision joint controllers.

Therefore Joint Positional control, in which the policy directly controls the rotation of all joints, seems to be the appropriate abstraction level considering the range of tasks and the robot that is used.

[3]https://developer.nvidia.com/physx-sdk
[4]https://www.universal-robots.com/media/1802780/ur3e-32528_ur_technical_details_.pdf

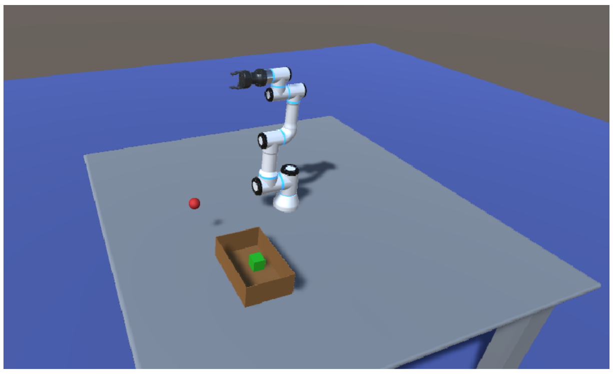


Fig. 3.2 Example Unity scene for the simulation environment.

As is common in robotics learning, the interpretation of the actions is a delta step on the current joint positions. It is the task of the embedded joint controllers to move the joints to these new target positions. Furthermore, as argued during the introduction, continuous action spaces are used as these are the research focus.

Another aspect related to the precision level is the control frequency. Higher control frequencies make it increasingly hard for the policy to learn the desired behaviors as it has to consider longer time horizons to accomplish the same tasks (a problem that Hierarchical RL tries to tackle) [27]. At the same time, this provides the policy with more feedback on the actions.

The Robotics environments of OpenAI use a control frequency of 25 Hz and hence this is also often used in recent work on joint positional control [28]. To reduce the task complexities, this environment uses a control frequency of only 10 Hz. This choice was not based on measurements[5] but only on intuition and visual inspection of the robot behavior for this control frequency (which did not seem to fail, so there was no obvious reason to increase the control frequency). For more fine-grained tasks such as pick and place of difficult objects, this control frequency likely needs to be increased to provide the policy with more feedback.

3.2 Unity Simulation World

The first major component that will be discussed is the simulation world. This Unity simulation world simulates the influence of the actions on the robot and its environment and reports back the state of the environment, as is shown in Fig. 3.1.

Each world (called a scene in Unity-speak) contains the robot and possibly some objects it can interact with, a box, some visual markers that indicate target positions, etc. An example scene is shown in Fig. 3.2. These additional objects are quite trivial and hence they will not be discussed in detail. The focus will be on the Robot and in particular on how to create joint position control.

[5]Note that no motivation for the 25Hz frequency could be found either, and its adaptation by the research community seems to be mostly motivated by inertia at first sight.

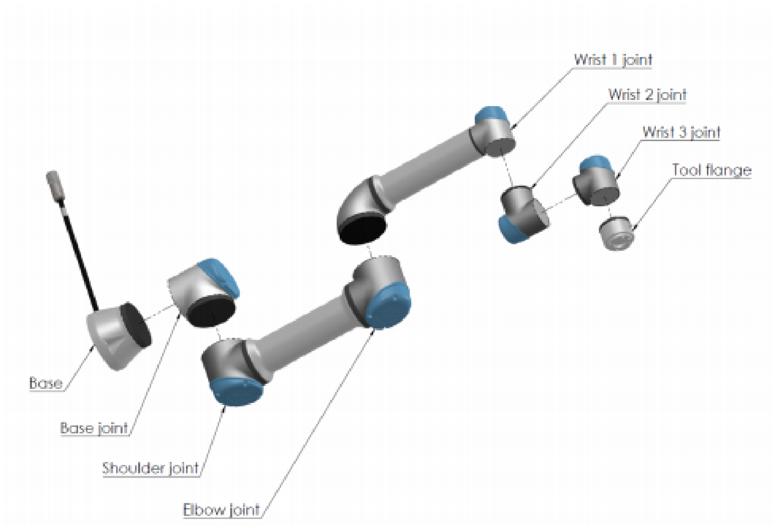


Fig. 3.3 Illustration[7]of the different degrees of freedom and corresponding joint names of the UR3e.

As stated before, the robot that was modeled is the Universal Robotics uR3e. This robot has 6 degrees of freedom, not counting the end-effector. The joints are indicated in Fig. 3.3. For the end-effector, the Robotiq Hand-E[6] parallel gripper is modelled.

Simulating a robot usually comes down to two tasks: Creating a 3D representation of the physical parts of the robot and how they can move relative to each other (i.e. making a model of the robot) on the one hand and creating the controllers that can make the robot move as required by converting the action inputs into actual movements on the other hand. Both parts are discussed next.

Robot Model

To increase the accuracy of the simulation, Unity's *ArticulationBody* components are used for the robot model. These articulationbodies enable the physics engine to simulate all robot joints in reduced coordinate systems using the Featherstone algorithm, hence making sure no movements take place in the locked dimensions of each link between the different parts of the robot [29].

The articulationbody components are chained together to create the articulation chain of the robot. Each component is anchored to its parent component and has a joint that describes how it can move with respect to this anchor. Different joint types exist, to express different relations between each component in the chain.

The model is based on a model provided by Unity in a demo project[8] for showcasing the articulation body component. However an error had been made with the joints of the UR3e, and a non-existing joint was added. This was corrected to make the joint configuration match the real UR3e as given in Fig. 3.3.

[6]https://robotiq.com/products/hand-e-adaptive-robot-gripper
[7]https://s3-eu-west-1.amazonaws.com/ur-support-site/41166/UR3e_User_Manual_en_Global.pdf
[8]https://github.com/Unity-Technologies/articulations-robot-demo

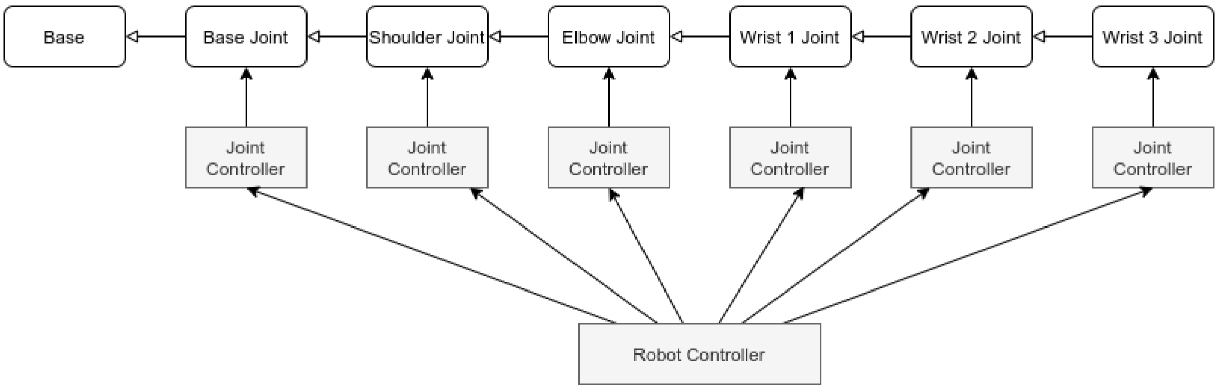


Fig. 3.4 Articulation chain (rounded boxes) and controller hierarchy (gray boxes) for the UR3e.

Robot Controller

To control the six Joints of the UR3e and the two joints of the parallel gripper, the following spring-damper equation (PD control) is used to calculate the torques/forces on each joint using Unity's *xDrive* components:

$$M = k(\theta - \theta_{target}) - c(\dot{\theta} - \dot{\theta}_{target}) \,, \tag{3.1}$$

where k is called the stiffness of the joint and c the damping. These are all set to default values of 1000 for k and 100 for c, which results in realistic movements of the robot manipulator.

On top of these low level drives a controller was created for each joint type that allows a.o. to limit the range of the joint, get the current position and velocity, update the target rotations, force the rotation to this target(to simulate a perfect controller) and reset the drives to an initial position. The home position of the robot, which is shown in Fig. 3.2, was for each joint chosen to be the 0 degrees orientation. Positive angles are defined by the right-hand orientation when pointing from the child to its parent.

All joint controllers of the robot and/or the gripper are grouped in a RobotController/ Gripper-Controller for ease of use. The resulting hierarchy for the robot can be seen in Fig. 3.4. For the robot, all rotations are specified in radians whereas the gripper uses a relative value between 0 (open) and 1 (closed) which allows controlling both joints with a single value.

Note that the robot controller was completely separated from the end-effector to allow for switching to a different end-effector if desired.

3.3 ML-Agents

The next high-level component of the environment is the ML-Agent. This agent has the following tasks, as indicated in Fig. 3.1:

- Initialize the robot and objects using the initial state provided to the agent.
- Set the environment goal as provided to the agent.
- Collect the required observations and the environment state.

- Send the observations and state to the next layer (the Python Interface), ask for a new decision (action) from the agent and pass this action to the robot and gripper controllers.

ML-Agents Framework

With ML-Agents [26], Unity provides a very convenient framework to manage the typical RL observe-decision-act loop. Although the framework contains an entire stack for designing and training intelligent agents, only the part that manages this RL loop and the API for communicating with the agent from python will be discussed as these are used for accomplishing the tasks that were described above.

An ML-Agent (as the ML-Agents Agent component will be referred to) has 3 main functions that perform the exact tasks that were outlined above. The *OnActionReceived* function simply sends the control inputs to the robot controller and gripperController. The *OnCollectObservation* function gathers all relevant information about the state of the environment and ,if required, additional sensor observations. The *onEpisodeBegin* function finally, uses the built-in Environment Sidechannel to get the initial state and the episode goal at runtime and uses this information to configure the environment at each episode. The ML-Agent can also deal with reward assignment and episode termination, but these functionalities are not used as both are handled in the python Interface for more flexibility.

A Unity environment can contain multiple ML-Agents, which are all managed by the *Academy*. This Academy keeps track of which agents need a new decision and will pause the unity world, collects observations and waits for a response from the python interface at each decision time.

Unity Time Management

As stated in Section 1, a decision frequency of 10Hz is desired for the agent. the ML-Agents framework provides a *decisionRequester* script that allows for requesting a decision from the agent every N steps of the physics engine, which takes regular steps according to an in-game time interval.

The physics step size was set to 0.01s in the environment, giving a 100Hz in-game physics frequency. Hence the *decisionRequester* should request a decision every 10 steps to obtain the desired 10Hz in-game control frequency.

The in-game time can then be scaled with respect to the realtime in order to speed up the simulations by scaling the physics frequency with respect to the wall clock time.

Finally, note that the ML-Agents framework by default attempts to lock the rendering frequency to the physics frequency. This is done to make sure that the visual observations that are collected by the agent, which are updated according to the rendering frequency, match the current state of the environment. This resulted in unnecessary rendering computations as there are no visual observations used and hence this lock was disabled.

3.4 Python Interface

The ML-Agents framework provides an interface based on the standard OpenAI gym template [30] for communicating in python with the simulator. This interface was extended to provide the following functionalities:

- Specify initial state and goal on reset of the environment.
- Communicate this initial state and goal over a sidechannel to the simulator at runtime.
- Externalize the reward calculation, as required for the goal-based MDP formulation that was introduced before to calculate the reward for arbitrary transitions and goals.

The resulting interface deviates from the OpenAI gym interface for goal-based environments [23] on two aspects:

- The observation is not a dictionary that contains the episode goal and the achieved goal. This was done to reduce the unnecessary overhead that is induced by communicating both goals at every step.
- The reset function takes as explicit arguments the initial state and the goal for the episode. This is required to vary the distributions over the initial state and goals, as is done in curriculum learning. The OpenAI Gym interface does not allow for this, as both initial state and goals are always randomly sampled in their goal-based environments.

3.5 Lessons Learned

Reflections on building a custom robotics learning environment

Building custom simulation environments for robotics has proven to be a time-consuming task. Furthermore, using a custom environment only increases the pool of possible issues that can result in a failure to learn the desired behavior. Next to wrong hyperparameter ranges, implementation errors in the learning algorithms and fundamental issues with the learning algorithms, issues with the learning environment are now also possible.

This additional source of possible issues combined with the implementation time illustrates the importance of having a suite of available environments and explains why almost all research on DeepRL uses pre-defined environments such as the OpenAI Robotics environments.

Applying DeepRL to real-world problems, however, requires designing tailored simulation environments as every situation has different requirements. Many simulation frameworks exist and support this process rather well. More guidance on common design choices (such as [31]) would help to speed up this process and result in appropriate abstraction levels, etc., which results in more efficient learning.

Reflections on the OpenAI gym interface

The OpenAI gym interface has become almost omnipresent in the DeepRL community. This makes it possible to easily swap algorithms on an environment or use the same algorithm on multiple environments.

Unfortunately, the gym interface source code is poorly documented and actually also not designed future-proof. The reset function of the base class for example does not take additional parameters, although this is required for any kind of curriculum learning where the distribution of goals initial states is varied.

This was one of the motivations to move away from this interface when creating the simulation environment. However, this has proven to be a mistake as it made interfacing with existing implementations difficult.

The better option would have been to stick to the gym interface as close as possible and to use the wrapper classes that are hidden in the codebase to further customize the environment interface to e.g. specify the initial states and episode goal, while still sampling them uniformly in the base class.

Chapter 4

Experimental Setup

This chapter gives an overview of the experimental setup that is used in future chapters. The first section introduces the different learning environments that are used. The next section gives a short overview of some tools and frameworks that are used.

4.1 Environments

In this section, the different learning environments that are used in this work are described. For each environment, a short description is given of the task. For the environments that were developed in this work, this is followed by the description of the state space, action space and goal space (if applicable) as well as how the actions influence the state. Finally, the initial state distribution(s) and reward function(s) are described. These elements make up the Markov Decision Process as introduced in Chapter 2.

4.1.1 Benchmark Environments

For testing and evaluating implementations of different DeepRL algorithms in Chapter 5, two environments from the OpenAI gym suite [30] are used. The OpenAI gym suite is a collection of often-used environments for Reinforcement Learning, introduced to make comparisons between algorithms easier and to introduce a standardized interface for interacting with RL algorithms. The environments that are used in this thesis are the Pendulum environment and the LunarLander environment. Both are 2D, single goal environments with continuous state and action spaces.

In the LunarLander environment, the agent has to learn to control the engines of a spacecraft and land it at a specific location, starting from the top of the game screen. The state space is 8-dimensional and is given by

$$\mathbf{s} = [x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, L, R],$$

where L and R are indicator functions that encode whether the left and right landing pad of the spacecraft are touching the ground. The action space is 2-dimensional, encoding the engine throttles. The agent is penalized for using the engines and rewarded for moving towards the landing pad. The agent receives an additional reward for landing and is punished for crashing.

The environment is considered to be solved if the agent achieves more than 200 points. The episode terminates if the spacecraft has landed or crashed, but an artificial maximum duration of 1000 steps is added.

The pendulum environment on the other hand requires the agent to balance a pendulum, starting from a random initial orientation and angular velocity. This environment has a 3-dimensional observation space given by $\mathbf{s} = [\cos(\theta), \sin(\theta), \dot{\theta}]$ and 1-dimensional action space encoding the torque on the pendulum. The reward is always negative and depends on the angular distance and velocity w.r.t. the desired position (which is upright). This reward function is continuous and convex, limiting explorational challenges. The environment is furthermore non-episodic, but as usual, an artificial maximum episode duration is introduced and set to 500. This environment has no threshold on the rewards that indicates the task has been solved, but the closer the reward is to 0, the better.

More details about these environments can be found in the repository for the OpenAI gym suite[1].

4.1.2 Pointmass Environment

This environment was developed to evaluate the asymmetric self-play framework on a low-dimensional, goal-conditioned task, which is done in Chapter 6.

The environment simply consists of a pointmass that can move around in 1 or 2-dimensional space. The state space is limited to $[-1.5, 1.5]^N$ and the actions are limited to $[-0.05, 0.05]^N$. Actions simply update the state as follows: $\mathbf{s}[t+1] = \mathbf{s}[t] + \mathbf{a}[t]$, implying positional control of the pointmass.

The goals $\mathbf{g}$ are defined as a point in the N-dimensional state space and are considered to be reached when the distance from the current state to the goal is less than 0.05, in which case the episode terminates. Mapping states to goals is trivial as they are equivalent.

The initial states can be either fixed to the origin or randomized over the entire state space. Rewards for the agent are binary: 1 if the goal is reached, else 0.

4.1.3 Motion Planning Environment

In Chapter 7, the asymmetric self-play framework is used to learn a robotics task using the Unity simulation environment that was introduced in Chapter 3.

In the learning environment constructed using this simulation environment, the agent has to learn to bring the end-effector of the UR3e Robot to a target position in the Euclidean space. Fig. 4.1 shows a particular state of the environment, where the target position is indicated with the red dot. This task can be seen as learning *Motion Planning* for the UR3e, which is a core task for robotics and is also a stepping stone towards unpacking items, the use case that was introduced before.

[1]https://github.com/openai/gym/tree/master/gym/envs

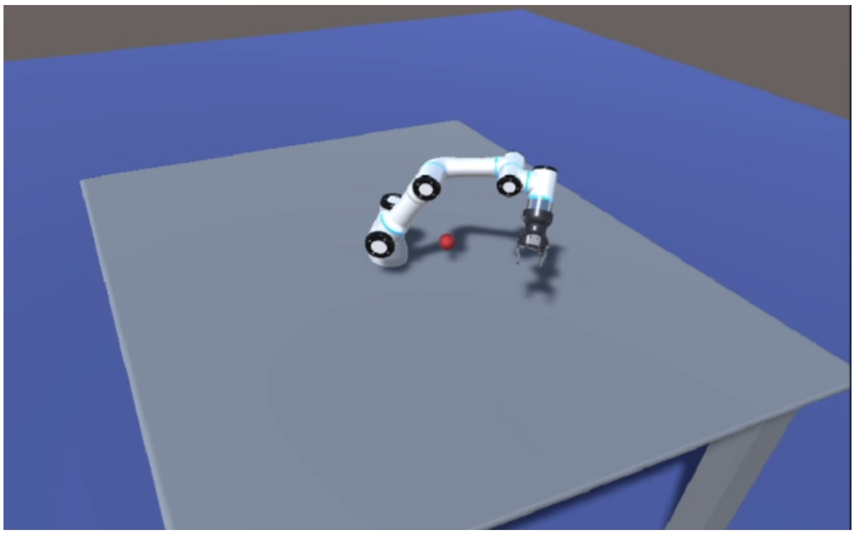


Fig. 4.1 Learning Environment for motion planning. The red dot indicates the position that needs to be reached by the end-effector.

Joint name	**Range**
base joint	[-90, 90]
shoulder joint	[0, 90]
elbow joint	[0,150]
wrist01 joint	[-180,180]
wrist02 joint	[-180,0]

Table 4.1 Joint space in degrees relative to the home position for the controllable joints in the motion planning environment.

Note that this is a basic environment, containing no obstacles whatsoever making it very well possible to perform this task using analytical methods. Furthermore, no collisions with the robot or the table are considered, which simplifies the task and environment design further.

The policy has to control 5 joints and bring the end-effector to a target position in the Euclidean space. Only 5 joints are used as the wrist03 joint (cf Fig. 3.3) and the gripper do not provide additional capabilities to the robot for this scenario.

The environment state consists of the current position $\boldsymbol{\theta} = [\theta_1, .., \theta_6]$ and velocity $\dot{\boldsymbol{\theta}} = [\dot{\theta}_1, .., \dot{\theta}_6]$ of all joints (including the one that is not controlled) of the robot, the relative gripper position[2] x_{gripper} and the Euclidean end-effector position $\mathbf{x} = [x, y, z]$. The joint space is given in Table 4.1. Including the end-effector position is a common choice in robotics learning problems, and does not limit real-world applicability if the dynamics of the robot are known (which is usually the case). The current velocity is included to make sure that the Markov property is satisfied for the state. The resulting state is 16-dimensional and given by:

$$\mathbf{s} = [\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}, x_{\text{gripper}}, \mathbf{x}] \,. \tag{4.1}$$

[2]The velocity of the gripper was not included in the state by mistake and this was only noticed later on as the gripper is not controlled in the motion planning environment.

The actions encode the desired relative orientation changes with respect to the current state:

$$\mathbf{a} = \boldsymbol{\delta}_\theta = [\delta_{\theta_1}, .., \delta_{\theta_5}] \, , \tag{4.2}$$

and are limited to an absolute value of 3 degrees or 0.052 radians, which gives a velocity of 30 degrees/s considering the decision frequency of 10Hz. The choice for 30 degrees/s is mostly arbitrary but the limitation serves to ensure the robot does not make unrealistically fast movements. These actions are then used to set the desired joint positions $\boldsymbol{\theta}_{ref} = \boldsymbol{\theta} + \boldsymbol{\delta}_\theta$ in Equation 3.1. The desired velocity $\dot{\boldsymbol{\theta}}_{ref}$ is always set to zero, as this task does not require the robot to control the velocities.

An artificial maximum episode duration is also added for the agent, although this duration can vary between experiments. This is because this duration influences the explorational capacities of the agent and hence there is a trade-off between learning speed and explorational capacity.

Initialization of the environment is done by selecting a random orientation for each joint within the joint space from Table 4.1.

The goal $\mathbf{g}$ represents a Euclidean point that has to be reached by the end-effector and is hence 3-dimensional. Extracting the achieved goal from the current state is done by setting the goal to the current end-effector position $\mathbf{x}$. Goals are considered to be achieved if the current end-effector position $\mathbf{x}$ is within 0.05m of the target position $\mathbf{g}$, in which case the episode terminates.

The goal space $\mathcal{G}$ is described by a quarter sphere with a radius between 0.1 and 0.7 m, and origin at the base of the robot. The quarter sphere is created by intersection with the plane of the table and the plane orthogonal on the table through the robot base, parallel with the front of the table. Note that the end-effector can also reach some positions out of this sphere. However, it is approximately the largest volume that can be reached given the joint space and can still be easily described and sampled from, without having to compute the forward dynamics explicitly.

Sampling this goal space is always done uniformly in the spherical coordinates with origin at the robot base:

$$r \sim \text{Unif}(0.1, 0.7), \phi \sim \text{Unif}(-\frac{\pi}{2}, \frac{\pi}{2}), \theta \sim \text{Unif}(0, \frac{\pi}{2}) \, , \tag{4.3}$$

and these are then used to compute the resulting Cartesian coordinates. This results in non-uniform sampling in Cartesian coordinates as this would require correcting for the the change in volume:

$$dV = dx \; dy \; dz = r^2 sin(\theta) \; dr \; d\phi \; d\theta \, . \tag{4.4}$$

This has an impact on the experiments in chapter 7, as the evaluation metric uses this distribution and hence for experiments with asymmetric self-play, not all goals would have equal importance in the curriculum. This mistake was unfortunately only caught after performing all experiments. It is expected that this issue does not impact any conclusions drawn from the experiments although it might influence the asymptotic performances slightly. The goal space and distribution are illustrated in Fig. 4.2.

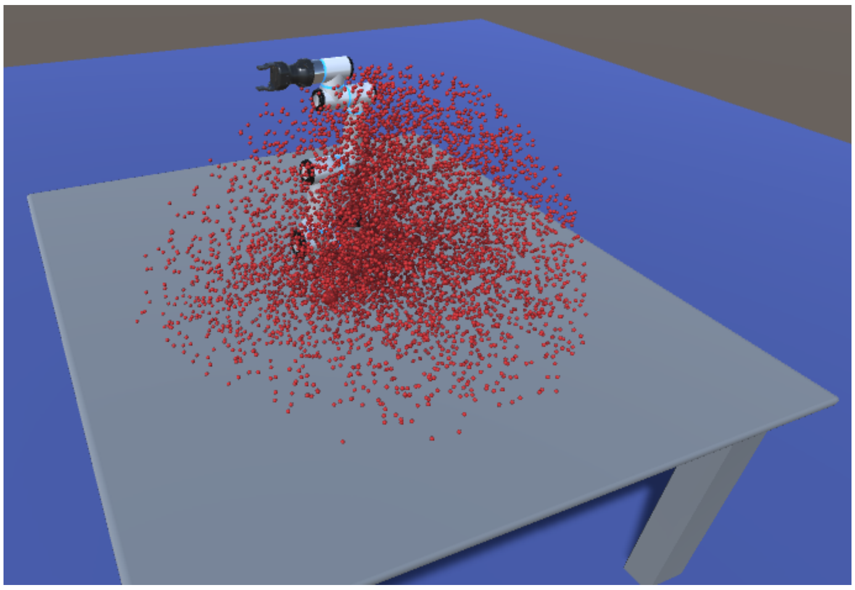


Fig. 4.2 Illustration of the goal space by sampling uniformly in spherical coordinates.

The reward function that is used can be either binary (1 if the goal is reached, 0 if not) or dense. This dense reward function is given by:

$$r(\mathbf{s}, \mathbf{a}, \mathbf{s}'|\mathbf{g}) = ||\mathbf{x}_{s'} - \mathbf{g}||_2\,, \tag{4.5}$$

which is the Euclidean distance between the goal and the current end-effector position.

In Chapter 3, the simulation timescale was already briefly discussed. This timescale allows for making the agent interact with the environment at a higher wall-clock frequency by speeding up the in-game time, which results in faster training times. Setting this timescale too high, however, could reduce the quality of the simulation physics, which results in undesired behavior of the agent. The upper limit of the timescale depends on the specifics of the learning environment and hence an experiment was performed to find this maximum allowable speedup. From this experiment, it became clear that a timescale of up to 20 did not result in a decrease in physics performance, although the runtime at this point was almost entirely dominated by non-physics code execution, which makes further increasing this timescale not useful. Hence a timescale of 20 is used for all experiments.

4.2 Tools, Hardware and Frameworks

All experiments were executed on a workstation with 2 Nvidia TITAN Xp GPUs and a GeForce GTX 1080 GPU, where the GPU was chosen randomly for each experiment. The workstation has an Intel i7-5820 CPU and runs on Ubuntu 18.04 LTS.

All code is written using the Pytorch framework [32]. A complete overview of all software dependencies and their versions can be found in the accompanying code repository[3]. This repository also includes the complete codebase used during this thesis.

[3]https://github.ugent.be/tlips/thesis-curriculum-learning

For managing the experiments and logging metrics, *Weights and Biases* (wandb) [33] is used. This tool allows for logging experiment results directly to a cloud-hosted database with an easy-to-use web client and API. It also provides tools for orchestrating distributed experiment sweeps (e.g. hyperparameter search or multi-seed experiments) and version management of datasets and models. It is mostly focused on Deep Learning research but also rather useful for performing DeepRL experiments.

Finally, Stable Baselines 3 (SB3) [34] is used to obtain a public implementation of some DeepRL algorithms. This framework provides well-documented and high-quality Pytorch implementations of many popular DeepRL algorithms. The implementations are explicitly benchmarked against the reference implementations of OpenAI Baselines [35].

Chapter 5

Learning Algorithms

In this chapter, the different DeepRL algorithms that are used throughout this thesis are introduced. In theory, these algorithms can be considered as background material and one could refer to their respective paper for details to implement them or simply use a publicly available implementation. However, the performance of these algorithms can be quite dependent on very small implementation details. Details that are often not (explicitly) mentioned in the original papers proposing these algorithms, or require combining elements from different papers. Furthermore, popular public implementations often use rather different combinations of these *tricks*.

It was Henderson et al. who brought these issues under the attention of the research community. They showed that performance can differ drastically between often-used public implementations and their implementation details, between commonly used network architectures and most notably between different sources of randomness such as the random seed generators or simply the GPU environment [36].

The goal of this chapter is hence twofold: First and foremost it aims to provide all the necessary implementation details of the algorithms that were used, and to validate these implementations. Secondly, it aims to provide insight into the characteristics of the algorithms that are used. It would be impossible to motivate the choice for particular algorithms for Alice and Bob in later chapters on asymmetric self-play, without having some insight into these characteristics.

In the first section, the mathematical foundations of Policy Gradient Methods are briefly stated before introducing a state-of-the-art policy gradient algorithm: Proximal Policy Optimization (PPO). The next section introduces a popular Q-learning algorithm: Deep Deterministic Policy Gradient (DDPG) with some extensions. Thereafter, the implementations of these algorithms are validated against a public implementation. Finally, some lessons learned during the implementation and testing of these algorithms are formulated.

5.1 Proximal Policy Optimization (PPO)

PPO is an on-policy, stochastic policy gradient method. Before formulating the algorithm and listing some of the tricks that are often used, the mathematical background for optimizing policy gradient methods is shortly given.

5.1.1 Policy Gradient Methods

As stated in Chapter 2, the goal of any RL learning algorithm is to maximize the expected (discounted) return, that is to maximize

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]\,. \tag{5.1}$$

Policy Gradient Methods try to optimize the policy π_θ directly by maximizing the gradient $\nabla J(\pi_\theta)$ using a gradient optimization technique.

It can be shown that this gradient can be expressed as

$$\nabla J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]\,. \tag{5.2}$$

This result is known as the Policy Gradient Theorem and is the mathematical basis of all policy gradient methods [14].

It can be also be shown that this can be formulated more generally as

$$\nabla J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t\right]\,, \tag{5.3}$$

where Φ_t can be a.o. the advantage function $A(s_t, a_t)$, which is defined as $Q(s_t, a_t) - V(s_t)$ [37]. Due to the lower variance, this formulation is almost always chosen in state-of-the-art policy gradient methods. It does however require an additional network that approximates the value function $V(s_t)$ to estimate this advantage from the rollout rewards.

5.1.2 Formulation

The motivation for the Proximal Policy Optimization (PPO) algorithm, as presented by Schulman et al., was to find a learning algorithm that allows for performing multiple epochs of optimization over each rollout while being stable and robust[1] with respect to the hyperparameter choices [38]. Iterating multiple times over the data is common practice in Supervised Learning to increase sample (data) efficiency, but had not been used for Policy Gradient Methods as it was empirically shown to often lead to destructively large policy updates.

Schulman et al. start from the Trust Region Policy Optimisation (TRPO) algorithm [39] and leverage the idea of limiting the step size that is allowed in each optimization step to make sure the policy does not move too far away from the current policy.

[1]as far as this goes in DeepRL

They formulated the following loss function for the policy network for a single step:

$$L_t^{CLIP}(\theta, \theta_{old}) = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}(s,a), \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}(s_t,a_t)\right). \quad (5.4)$$

If desired, an entropy measure $S(\pi)$ of the stochastic policy can also be added to help regulating the variance of the policy, which will tend to decrease over time as the policy converges.

As discussed before, the advantage estimation requires learning an approximation of the value function $V(s)$. In DeepRL, this approximator is a neural network and is often made to share all but the last layer with the policy network. The idea here is that both networks should learn a similar representation of the observations. The loss function hence also includes the value loss since the same optimizer is often used for the partially shared policy network and value network.

The final objective then looks as follows:

$$L(\theta) = \mathbb{E}_t\left[L_t^{CLIP}(\theta, \theta_{old}) - c_1 L^{VF}(\theta) + c_2 S(\pi_\theta)\right], \quad (5.5)$$

and is estimated using a finite number of rollout steps and then maximized at every iteration with a (batched) optimizer for a number of epochs.

To estimate the advantages, Schulman et al. propose to use the Generalized Advantage Estimation (GAE) [40]. This method trades an acceptable amount of bias on the estimation to reduce the variance significantly. Furthermore, they perform an additional modification by estimating the return of the final state with the value function to allow for having rollouts that do not contain terminated episodes or for non-episodic environments that do not even have a natural termination state, as proposed in [41].

The advantage estimation is given by:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \quad (5.6)$$

$$\text{where } \delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t).$$

As stated by the authors of GAE, for $\lambda = 1$ this reduces to

$$\sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'} + V_\phi(s_T) - V_\phi(s_t), \quad (5.7)$$

which is simply the Monte Carlo advantage estimation using the future rollout rewards. The parameter λ allows for trading off the high variance of this formulation (due to the summation) with a non-zero bias of the estimation [40].

The value loss is computed using the mean squared error over the return estimations $\hat{R}(s)$:

$$L^{VF} = \left(V_\phi(s) - \hat{R}(s)\right)^2 . \tag{5.8}$$

The resulting algorithm can be found in Appendix A.

This algorithm outperforms most other stochastic policy gradient methods in terms of efficiency, asymptotic performance, stability and hyperparameter robustness and is hence considered to be the go-to on-policy stochastic policy gradient algorithm.

Finally, note that Schulman and Al. also proposed a variant of the PPO algorithm as discussed above in which the policy gradient steps are limited by a Kullback-Leibler (KL) penalty (PPO-KL) instead of being clipped. This version however showed slightly worse performance and hence the clip method (referred to as PPO-CLIP) is usually preferred.

5.1.3 Additional Modifications

A closer look at the Stable Baselines [34] implementation revealed that they performed a number of additional modifications to the algorithm as proposed by Schulman et al.:

- The gradients of the network are clipped before optimizing them.
- The batches are reshuffled at each epoch iteration.
- The advantage estimations are normalized over each batch.
- The value loss is computed using the advantage estimations instead of the Monte Carlo value estimations.

The impact of these modifications is discussed in more detail in Section 5.3.1.

5.2 Deep Deterministic Policy Gradient (DDPG)

DDPG differs from PPO in almost all aspects. First of all, it is an off-policy algorithm which means that samples can be collected using any policy. Furthermore, it is a Q-learning algorithm in which the central learned function represents not the policy itself but the Q-function. DDPG does have a second network that represents the policy but this is only trained indirectly on the Q function as will become clear soon.

5.2.1 Formulation

The DDPG algorithm combines some of the tricks of the DQN algorithm (which was the first algorithm that managed to use neural networks as function approximators for Reinforcement Learning in high-dimensional observation spaces) [17] with the Deterministic Policy Gradient method [42].

The central equation for all Q-learning algorithms is the Bellman Equation, given by (for a deterministic policy):

$$Q(s,a) = \mathbb{E}_{s' \sim P}[r(s,a) + \gamma Q(s', \mu(s'))]\,, \tag{5.9}$$

where $\mu(s)$ is the greedy policy defined by

$$\mu(s) = \text{argmax}_a Q(s,a)\,. \tag{5.10}$$

Note that this equation holds for all transitions (s, a, r, s', d) of the MDP and hence does not require them to be obtained using the current policy (which explains why DDPG is a so-called off-policy algorithm). Therefore a replay buffer $\mathcal{D}$ is usually introduced in which a number of previous transitions are stored and from which at each optimization step a random batch is sampled.

Since DDPG is used on continuous action spaces, one cannot easily compute the argmax in Equation 5.10. Therefore, the greedy policy is usually approximated by a second network $\mu_\theta(s)$ [37], which is optimized by maximizing

$$\mathbb{E}_{s \sim \mathcal{D}}\left[Q_\phi(s, \mu_\theta(s))\right]\,. \tag{5.11}$$

To optimize the Q function parameterized by ϕ, following loss function[2] needs to be minimized:

$$L(\phi) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}}\left[\left(Q_\phi(s,a) - (r + \gamma(1-d)Q_\phi(s', \mu_\theta(s')))\right)^2\right]\,. \tag{5.12}$$

This loss function $L(\phi)$ however, depends twice on the network Q_ϕ, which makes the optimization 'chasing its own tail' and leads to unstable behavior [43].

To reduce the effect of this double dependency, next to using a large replay buffer to limit the sample dependencies and clipping the gradients, an additional trick is used: a so-called target network is introduced for both the Q-network and policy network. This target network is used for computing the second half of the Bellman Loss. The target networks are updated at each iteration by a weighted average of the network parameters, often referred to as Polyak updating:

$$\phi_{\text{target}} \leftarrow \tau\phi + (1-\tau)\phi_{\text{target}}\,. \tag{5.13}$$

Although the algorithm is off-policy and hence any sampling strategy could be used to obtain new samples for the replay buffer, usually, a noisy variant of the current (deterministic) policy is used as this leads to more informative samples in general. Lillicrap et al. propose to use correlated noise according to an Ornstein-Uhlenbeck process, but later work has found (decaying) Gaussian noise to work just as well and hence the latter is usually preferred due to its simplicity [37].

[2]Here I followed the OpenAI Spinning Up notation that uses python-like evaluation of the terminal state boolean value

5.2.2 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Whereas Policy Gradient methods often suffer from high variance issues, Q-learning-based methods tend to suffer from *overestimation bias*. This is caused by recursively maximizing over the inherently noisy value estimations of the function approximator, which might lead to high values for certain state-action pairs resulting in suboptimal policies or even complete divergence of the policy [44].

To address these overestimation issues and in an attempt to reduce the brittleness of the DDPG algorithm in general, Fujimoto et al. propose three distinct updates to the DDPG algorithm, partially inspired by earlier updates to the DQN algorithm [45]:

1. Instead of learning a single Q-function, two Q-functions are learned separately. During the computation of the Bellman targets, the lowest of the two Q values is used.
2. Gaussian noise is added to the target actions for the sampled states from the buffer. This is referred to as target policy smoothing. The noise is clipped to stay close enough to the actual target.
3. The policy network is updated less frequently (once every n updates of the Q networks)

The resulting algorithm can be found in Appendix A.

TD3 is considered to be state-of-the-art for continuous control, beating both PPO and DDPG on a suite of OpenAI gyms [44].

5.3 Evaluating the Implementations

In this section, the implementations as described previously are benchmarked against Stable Baselines [34] using the LunarLander environment that was introduced in chapter 4.

For these tests, all hyperparameters are reported as accurately as possible and the random seeds of the different libraries are fixed to the same value before each run. As the goal is not to assess absolute performance but rather to perform what could be considered an *integration test* on the implementations, performance is not averaged across multiple seeds.

The reported performance metric is the non-discounted cumulative episode reward during training. The stable baselines implementation averages these rewards over all previous episodes, whereas this implementation simply reports the average episode reward of each rollout, which is then averaged using a rolling window of size 10. This choice was made because it seems that not averaging over all previous episodes should allow to better observe stability issues etc during training.

5.3.1 Evaluating PPO

Influence of Modifications

As indicated, the stable baselines implementation, as well as others, adds some modifications to the original algorithm. Reshuffling the batches and clipping the gradients were included in the

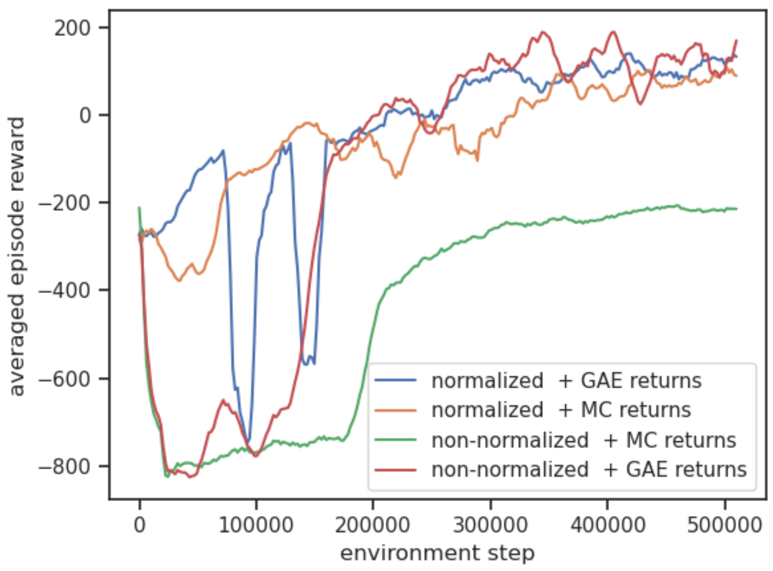


Fig. 5.1 Comparison of PPO performance for normalized Advantages and different estimation methods for the Returns.

implementation as these are common tricks to improve stability, although they did not appear to make too much difference.

The different estimations for the returns and the batch normalization appeared to be more important. Most notably when using Monte-Carlo (MC) return estimations combined with non-normalized GAE advantages (which would be the default if you only looked at the paper), the algorithm was not able to obtain a high performance the LunarLander environment, as can be seen in Fig. 5.1. The final implementation uses GAE estimations to compute the returns and normalizes the advantages.

Testing the Final Version

To validate the resulting implementation, the algorithm was tested on the LunarLander environment and the performance was compared to the StableBaselines3 implementation. The hyperparameters were chosen identical to the default values in the SB3 implementation, which correspond to the suggested values of the original paper. These values are listed in Table 5.1.

Note that the network architectures for the action mean differ slightly: Both networks consist of two hidden layers that are shared between the policy and value network. However, the SB3 implementation uses tanh as activation function whereas this implementation sticks with ReLU due to bad experiences with tanh, which seems to require more normalization tricks on the inputs of the network. Furthermore, this implementation uses a hidden layer size of 128 and uses the default Pytorch initialization, as always in this thesis, whereas SB3 uses 64 hidden nodes per layer and orthogonal initialization of the weights.

Hyperparameter	Value	Hyperparameter	Value
epochs	10	rollout lenght	2048
batch size	64	GAE factor	0.95
learning rate	1e-4	gradient clip norm	0.5
value loss coeffficient	0.5	entropy loss coefficient	0.0
ppo clip value	0.2	discount factor	0.99

Table 5.1 Hyperparameters for the PPO Benchmark.

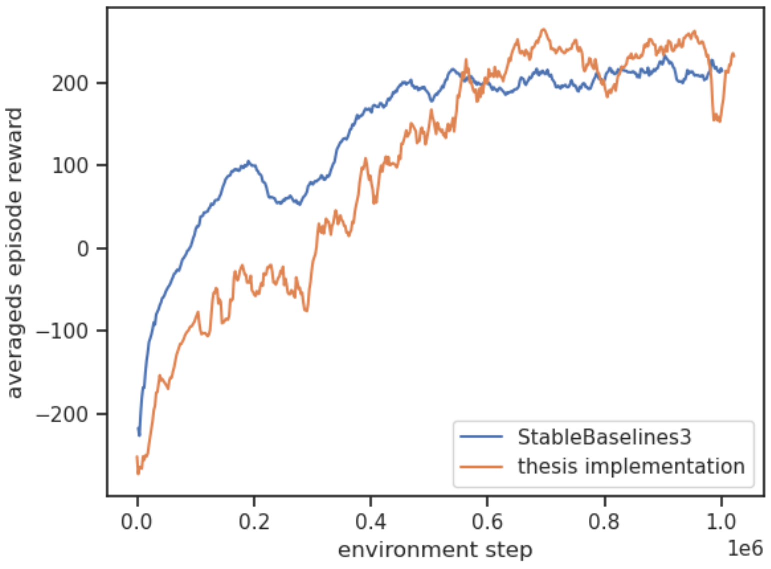


Fig. 5.2 Benchmarking the PPO implementation on the LunarLander-v2 environmnent.

In both implementations, the diagonal standard deviation of the policy is represented by a state-independent vector, that learns the natural logarithm of the standard deviation. This vector is initialized to zeros, resulting in an initial standard deviation of 1.

Results can be found in Fig. 5.2. The performance, in terms of asymptotic cumulative reward and learning efficiency, of both algorithms is rather close which suggests the implementation has no issues that would hurt performance too much.

5.3.2 Evaluating DDPG

The implementation is again compared to the StableBaselines3 implementation on the LunarLander environment with hyperparameters as listed in Table 5.2. The StableBaselines implementation

Hyperparameter	Value	Hyperparameter	Value
actor learning rate	1e-3 / 1e-4	critic learning rate	1e-3
buffer size	1e6	batch size	128
train frequency	1 / step	noise	$\mathcal{N}(0;0.1)$
gradient clip norm	1.0	discount factor	0.99

Table 5.2 Hyperparameters for the DDPG Benchmark.

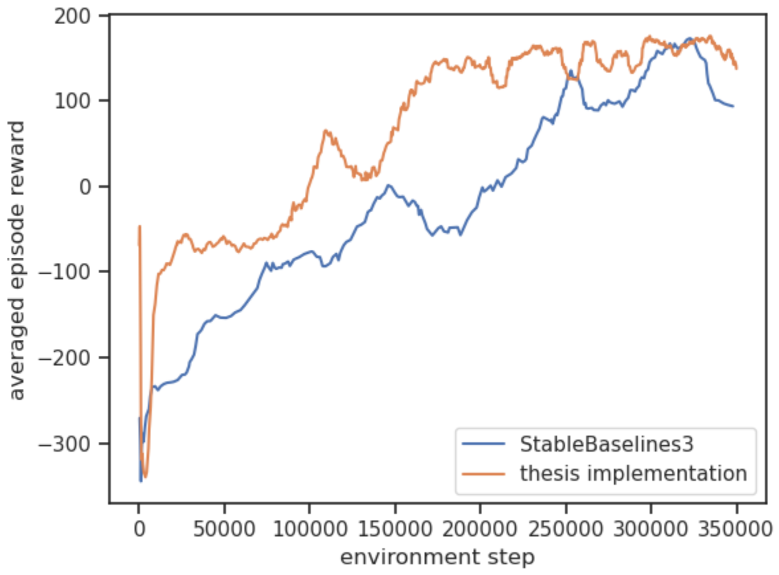


Fig. 5.3 Benchmarking the DDPG implementation on the LunarLander-v2 environment.

does not allow for selecting a different learning rate for the value network and policy network so both were set to the default value of 0.001. This implementation uses the same value for the Q-network but a lower value of 0.0001 for the policy network as this is what is used in the original paper [13].

Again the networks differ in architecture as the SB3 implementation uses hidden layers of size 400 and 300 whereas this implementation uses 128 for both layers. The policy and Q-network do not share the hidden layers.

The results can be seen in Fig. 5.3, where again the implementation seems to be on par with the reference framework, suggesting that there are no large problems with the implementation.

Note that the asymptotic reward is lower than for PPO, which is most likely due to non-decreasing noise that is added to the actions. This is in contrast with the decrease in the standard deviation of the PPO policy, as the algorithm is converging. This leads to more "qualitative samples" on the one hand but mostly limits the disturbance during rollouts on the other hand, explaining higher rewards.

To increase the quality of the DDPG samples, one could use a decreasing standard deviation for the action noise. However, the DDPG algorithm tends to be very brittle with respect to tuning these noise parameters. Furthermore keeping the noise higher has the benefit that the learned policy is more likely to be robust as this noise could be seen as some kind of adversarial perturbation.

However, for more fine-grained task such as reaching for items in a small box, high noise is prohibitive as it would hinder the policy from reaching the item without touching the box. Exploration issues with such fine-grained control tasks are hence even more challenging.

Hyperparameter	**Value**	**Hyperparameter**	**Value**
learning rates	1e-3	discount factor	0.99
buffer size	1e6	batch size	128
train frequency	1 / step	action noise	$\mathcal{N}(0;0.1)$
target noise parameter	0.2	target noise clip value	0.5
policy update frequency	2		

Table 5.3 Hyperparameters for the TD3 Benchmark.

To avoid the influence of noise on the performance metrics, one can perform separate evaluation rollouts where no noise is added to the actions. This is not done here as the goal was mainly to test the implementation rather than to obtain absolute performance measures.

Visual observation[3] of the learned policy shows that the policy does indeed learn to land, even though it does not reach the score of 200 due to the added noise as explained above.

Finally note that the required number of interactions before reaching the asymptotic performance is indeed smaller with respect to the PPO algorithm, highlighting the increased sample efficiency of off-policy algorithms. However in terms of computational wall clock time, the DDPG algorithm performed worse as the networks are updated more frequently, something which is of course highly dependant on the hardware platform.

5.3.3 Evaluating TD3

As before, the resulting implementation is compared to the SB3 implementation. The hyperparameters are listed in Table 5.3. As was the case for DDPG, the network architectures differ: whereas the Stable Baselines implementation uses hidden layer sizes of 400 and 300, this implementation uses 256 for both layers.

The results can be found in Fig. 5.4. Note that there are two line plots next to the baseline. This is because during the writing of this manuscript after all experiments had been performed, an implementation error was discovered in the TD3 implementation: The noise added to the target actions in the implementation is Uniform instead of Gaussian. To briefly evaluate the impact, the LunarLander test has been performed with Uniform and Gaussian target noise. As can be seen, the impact of this bug seems to be not too large. Hence it is expected not to influence later experimental results too drastically, although it is still important to mention for completeness. The bug has also been added to the algorithm in Appendix A.

Another observation is that the stable baselines implementation reaches a higher performance than the implementation, although the algorithms should have been allowed to run a little longer to make sure the asymptotic performance was reached. This difference is most likely caused by additional implementation differences[4].

[3]https://youtu.be/LzPReCrLXIM

[4]There was unfortunately no time left to investigate this further as the TD3 algorithm was initially only benchmarked on the Pendulum environment, where no performance gap was observed. During the writing of this manuscript, it appeared that using the same environment for all algorithms would be nice, and only then this difference was discovered, as well as the target action bug.

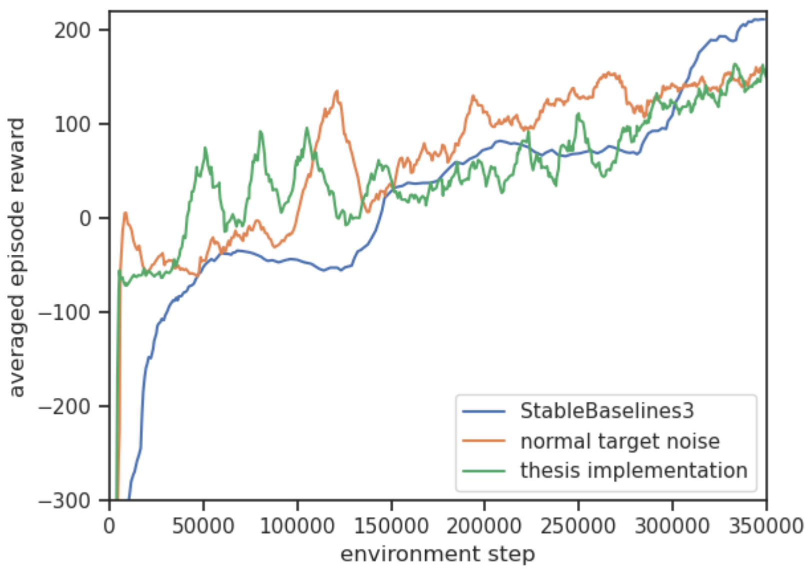


Fig. 5.4 Benchmarking the TD3 implementation on the LunarLander-v2 environment.

5.4 Lessons Learned

In this section, some of the lessons learned during the implementation and testing of these algorithms are briefly stated.

Public implementation vs. own implementation

At first, it felt strange to re-implement existing algorithms. However as already explained, details matter, and implementing an algorithm helps to understand all details better. Furthermore, qualitative public implementations are often embedded in a software architecture, which implies one cannot simply extract the code for a particular algorithm. Simply using the framework interface on the other hand makes it hard to build on these algorithms (as is done later in this thesis), often reduces the flexibility to adapt logging metrics and limits the option to change implementation details, etc. So it seems that creating a custom implementation is indeed the best option for doing research. This is however not straightforward as DeepRL algorithms are very brittle and can be rather complex. Making them work, even on very simple environments, is far from trivial. Having a look at the code from well-validated implementations, both before and after reading the original paper, will speed up this process, although it remains time-consuming.

Choosing the right benchmark environment

In this chapter, two different environments were used initially, where the pendulum environment is considerably easier than the LunarLander as it has a convex optimization landscape and lower dimensionality. This allowed for faster iterations during development. On the other hand, this also occluded some bugs that were present in the algorithm (including comparing scaled and unscaled actions in the value loss and the difference in performance for the TD3 algorithm, which was not visible with the pendulum environment). This is why the more difficult LunarLander was eventually used.

Testing an algorithm should preferably be done on an environment of significant complexity, although even then there are no guarantees. Making 100% sure that the implementations are working fine, requires testing on a whole suite of environments but in practice, this is often too time-consuming.

Interaction loop matters

Most learning algorithm papers focus on the training loop, which is of course where the novelty of the proposed solutions lies. However, when implementing algorithms one should not overlook the importance of the interaction loop in which the experiences are collected. Two particular bugs that required quite some time before stumbling upon them were:

- Handling artificial terminations of non-episodic environments, in which case one should not mark the transitions as *done* to avoid confusing the value networks.
- The scaling of actions from the [-1,1] range that is produced by the tanh in the final network layer to the ranges required by the environment. This should either be done consistently in the network itself, or one needs to take care not to scale the actions before adding them to the experience buffers and to only scale them before sending them to the environment. Otherwise, during the training it could be that non-scaled actions are used for some updates and scaled actions for others, which results in a complete failure of the algorithm.

Hyperparameters matter, bugs even more

It is well known that hyperparameters matter a lot in Deep Learning and this is arguably even more true in Deep RL. It was hence tempting to start tweaking hyperparameters if the algorithm did not manage to learn as expected during testing. However, there is one thing that has a greater influence on the learning algorithms: bugs. Hence when implementing a learning algorithm a good strategy is to search in the original paper for the hyperparameters that the authors have reported to work well on similar environments. Using these hyperparameters the odds become even bigger that the algorithm not working is due to implementation issues. Alternatively, one could first experiment on the environment using a public framework and then take the same hyperparameters. A small caveat here is to make sure that the interpretation of all hyperparameters is as expected, for example when using the OpenAI Spinning Up implementation of DDPG switches the weights of Equation 5.13, resulting in catastrophic instabilities when using the same Polyak hyperparameter value.

Get to know the algorithm

Even though initially one should think more about bugs than hyperparameters, it is worthwhile not to jump to the next task immediately after getting the algorithm to work on a test environment. Evaluating the algorithm's sensitivity to some of the hyperparameters can be very insightful and help to analyze issues later on as well as setting sensible default ranges for some hyperparameters in later parameter searches. Unfortunately, I did not do this, as I was too eager to start working on the next item on the TODO list. Looking back I believe it would have been an efficiency gain in the long run though. To make this process more efficient, there exist some papers like [46], where the influence of hyperparameters is discussed. This can be a useful guideline.

Loss function interpretation

Many loss functions have been formulated in this chapter. However, interpreting loss functions in DeepRL should be done with caution. In supervised learning, the loss function measures performance over a fixed data distribution, independent of the parameters that are being optimized. In DeepRL however, loss functions do not necessarily measure performance and do most certainly not measure over a fixed data distribution as the policy is typically used for data collection [37]. As a consequence, loss functions do not behave as in supervised learning and they do not necessarily need to be monotonically decreasing to indicate that the training process is going well. Measuring performance and stability should be done using the cumulative rewards.

Chapter 6

Asymmetric Self-Play

In this chapter, asymmetric self-play is discussed in more detail. The chapter starts by formulating some issues that can arise due to the interactions between the two agents in the framework. The next section covers the implementation of the actual framework and relates some implementation choices to the aforementioned issues. The following two sections describe different possible reward structures for Alice and some methods for increasing the diversity in the goals presented to Bob, which will prove to be the main issue of the framework. In the next section, these reward structures and different modifications are evaluated on a 1 or 2-dimensional point mass environment. The last section discusses the results of these experiments and draws some initial conclusions on the use of asymmetric self-play with continuous action spaces.

6.1 Possible Issues with Asymmetric Self-Play

As the asymmetric self-play framework involves interactions between two agents, it has a number of potential issues. Some of these were already brought up in Chapter 2 as they are already mentioned in other work related to goal curriculum learning. All of these issues can manifest themselves in both discrete and continuous action spaces, although some are more severe with continuous action spaces because of the unimodal policy representation.

The potential issues are briefly described below:

Alice > Bob

This problem relates to Alice exploring the goal region without taking care of presenting goals to Bob that are not too hard to reach, given his current capabilities. This can result in too many non-informative experiences for Bob with sparse or delayed rewards, as the probability of reaching these goals is simply too low.

Bob > Alice

Imagine Bob has learned to perform all tasks Alice is proposing at a certain moment in time quite well. Then it could be that Alice has no guidance on how to explore the goal space further as Bob manages to solve all tasks in the nearby space (which Alice would be able to reach given her explorational capacities). Hence Alice would most likely not be able to find the remaining goals that Bob has not yet learned to solve. This would limit the generalization of Bob as he would most likely not be able to solve all goals in the goal space since he was only presented with a subset of goals.

This issue was briefly mentioned by the original authors [1] and was also described by Florensa et al. [12].

Note that this problem is more likely to occur in combination with the next issue that is described, although it could in theory also take place if Alice for example fails to learn more difficult goals fast enough.

Convergence of Alice

This refers to the main expected issue with Alice in continuous action spaces and is related to how Alice represents her action policy. In discrete action spaces, the policy usually learns a categorical distribution over the action space. This categorical distribution is multimodal by nature and hence would allow Alice to learn to reach multiple goals at the same time, which she can then present to Bob. In continuous spaces, however, policies usually learn a Gaussian distribution over the action space, which is a unimodal distribution. This implies that in this case, Alice would only be able to learn one direction in the goal space at a time from which she can present goals to Bob.

This is by itself not a fatal issue as Alice could still learn to cover the entire goal space, though it would take considerably longer. However, it will often lead to either the problem described before (Bob > Alice) or the one described next (Bob overfitting), which will result in the self-play framework getting stuck. This unimodality issue was also mentioned by Florensa et al. [12].

Overfitting of Bob

Suppose Alice would only present Bob with a subset of the goal space for a given number of episodes. It would then be possible for Bob to actually overfit on this subspace, which will likely be easier to achieve than the generalized task space Alice is supposed to teach Bob. Even if Alice were to later on move to a part of the goal space, it might prove very hard for Bob to undo the overfitting that has taken place as this will typically require a great deal of explorational power.

Note again that this problem is more likely to occur in the case where Alice is converging, but that it is not strictly limited to it.

6.2 Asymmetric Self-Play Framework

In this section, a version of the asymmetric self-play framework is developed and its implementation is discussed.

From the components and interactions of Fig. 6.1 it is clear that four decisions need to be made:

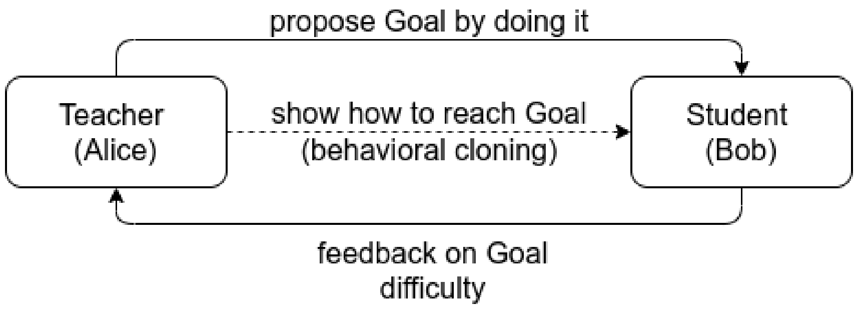

propose Goal by doing it
Teacher (Alice)
show how to reach Goal (behavioral cloning)
Student (Bob)
feedback on Goal difficulty

Fig. 6.1 Schematic illustration of the agents and their interactions in the asymmetric self-play framework.

1. Which learning algorithm to use for Alice,
2. which learning algorithm to use for Bob,
3. how to encode goals from Alice to Bob,
4. how to provide Alice with feedback on the goal difficulty, i.e. what reward structure to use for Alice.

The first three are discussed in this section, as well as Behavioral Cloning (BC), which was added to the framework by OpenAI [11], and is indicated with the dashed arrow in Fig. 6.1. Different reward structures to encode the feedback are discussed in the next section.

Learning Algorithm for Alice and Bob

The original asymmetric self-play algorithm, used Policy Gradient methods for both Alice and Bob [1], as does OpenAI [11]. As explained in Chapter 5, policy gradient methods require on-policy data collection and hence do not allow replaying transitions, which makes them less sample efficient.

In the context of asymmetric self-play an additional argument can be made in favor of using an off-policy algorithm for Bob: Taking into account the expected issues with the unimodal policy representation Alice will learn, the replay buffer can also serve as a way of presenting Bob with more diverse goals during the training by replaying a random batch of all goals Alice has ever proposed. Therefore, Bob's policy will be trained using the TD3 algorithm. Since a goal-based RL formulation is used, Bob's policy and Q networks take as input the concatenated state and goal $\boldsymbol{s}||\boldsymbol{g}$, as explained in Section 2.1.2 on multi-goal RL.

Alice on the other hand, cannot make use of such an off-policy method since her optimization landscape is constantly changing as Bob evolves, making on-policy data collection necessary. Therefore PPO is used for Alice since this is known to be a relatively stable method. As is done in the original formulation [1], Alice is given as input the current state as well as the initial state of the episode: $\boldsymbol{s}||\boldsymbol{s}_0$ to provide her with information on the trajectory so far.

For both PPO and TD3, the algorithm and all implementation details were described in Chapter 5. Pseudo-code for the implementations can be found in Appendix A.

Communicating Goals to Bob

In the original ASP framework [1], Bob's goals during self-play simply consisted of the final state of Alice, which was passed by Alice to Bob. As this final state was not part of the (single-goal) goal space of the task, Bob was given an artificial time-based reward during self-play episodes instead of using the environment reward signal $r(s, a)$.

The introduction of the goal-based extension of the RL formulation (cf. Section 2.1.2) removes this need for an artificial reward signal. For every state Alice reaches, a corresponding goal can be extracted using the state to goal mapping. This goal is then presented to Bob, as would be done in non-curriculum training using random goals. Hence Bob can simply be rewarded with the environment reward signal during self-play. This formulation was also used by OpenAI [11].

Behavioral Cloning (BC)

An addition to the original framework is inspired by the work of OpenAI [11]. As referred to in Chapter 2, one benefit of using a second agent to present goals by reaching them, is the certainty that there exists at least one trajectory to achieve the goal, namely the trajectory Alice used to get from the initial state to the goal state.

Using this trajectory one could actually perform some kind of Behavioral Cloning (BC) on Bob. However as always with BC, Bob's performance will be limited by that of Alice. Alice does not have as a learning objective to solve the tasks, she only wants to find difficult goals for Bob. This could limit the quality of the trajectory w.r.t. Bob's objective. Thereto the OpenAI team suggested using Alice's trajectory only in case Bob did not manage to solve the goal within his maximum episode duration. Even then, they found that using BC potentially leads to instabilities and used PPO-style loss clipping and an additional hyperparameter to reduce the influence of the BC-loss.

This implementation, however, uses an off-policy learning algorithm for Bob which allows to combine Bob's own experiences with the trajectories provided by Alice in a single replay buffer.

Note that Behavioral Cloning is not always used, it is considered an addition to the framework.

Resulting Pseudo-Code

In Algorithm 1 high-level pseudo-code for the resulting asymmetric self-play loop is given for a general reward structure for Alice (this will be discussed in more detail in the next section). Algorithms 2 and 3 provide details about the interaction loops for Alice and Bob. No details are provided on the training steps for Alice and Bob as this was described in Chapter 5. Pseudo-code for these training steps is provided in Appendix A.

Note that for simplicity, it is assumed that Alice and Bob cannot take invalid actions. Hence, no check is performed during the interaction loop to ensure the last action did not result in a terminal transition due to invalid actions. The only way for a transition to be terminal, is if the goal was achieved for Bob.

Algorithm 1 Asymmetric Self-Play

```
Initialize A: Alice's PPO policy, B: Bob's TD3 policy, E: Environment, $\mathcal{D}$: Bob's replay buffer
for N episodes do
    Get initial state $s_0 \sim \rho_0$
    $g, \tau_A \leftarrow$ ALICE_LOOP($s_0$)
    $\tau_B \leftarrow$ BOB_LOOP($g, s_0$)
    $r_A \leftarrow$ ALICE_REWARD($\tau_A, \tau_B, g$)
    if Behavioral Cloning active and Bob did not reach goal then
        $\tau_{BC} \leftarrow$relabel $\tau_A$ with $g$
        Add $\tau_{BC}$ to $\mathcal{D}_B$
    end if
    Set rewards of $\tau_A$ to $(0, ..., 0, r_A)$
    Train Alice using $\tau_A$          ▷ see Appendix A for training details
    Train Bob using $\mathcal{D}$          ▷ see Appendix A for training details
end for
```

Algorithm 2 Alice Loop

```
Reset E using $s_0$ and dummy goal $g$
$t_A = 0$
while $t_A < T_{max,A}$ do
    Observe $s$
    Get noisy action $a \sim \pi_A(\cdot|s||s_0)$
    if $a$ is STOP then          ▷ depends on the reward structure if Alice can signal STOP, see section 6.3
        break
    end if
    Execute a in the environment
    Observe $s', r, d$
    Add transition $(s||s_0, a, 0, s'||s_0, d)$ to $\tau_A$
    $t_A = t_A + 1$
end while
$g \leftarrow$ extract $g$ from $s$
Return $g, \tau_A$
```

Algorithm 3 Bob Loop

```
Reset E using $s_0$ and Alice's goal $g$
$t_B = 0$
while $t_B < T_{max,B}$ and g not reached do
    Observe $s$
    Get noisy action $a = \pi_B(s||g) + \epsilon$, $\epsilon \sim \mathcal{N}$
    Execute a in the environment
    Observe $s', r, d$
    Add transition $(s||g, a, r, s'||g, d)$ to $\tau_B$
    $t_B = t_B + 1$
end while
Add $\tau_B$ to the replay buffer $\mathcal{D}$
Return $\tau_B$
```

6.3 Reward Structures for Alice

In this section, different reward functions for Alice will be formulated and some hypotheses on their effectiveness will be formulated w.r.t the issues described in Section 6.1.

Sparse Rewards

In their paper on goal-discovery with asymmetric self-play, the OpenAI researchers introduced sparse rewards for Alice [11]. If there exist no invalid goals for Alice, which is indeed assumed throughout this chapter, the reward for each episode of Alice might be simply given by:

$$r_A = \mathbb{1}_{\{\text{Bob did not reach the goal}\}} \,. \tag{6.1}$$

It should be immediately clear that such a reward structure will most likely suffer from the A>B issue since Alice's incentive is simply to find goals Bob cannot achieve.

A possible modification is to introduce an external mechanism to regulate the goal difficulty according to Bob's current capabilities and will be referred to as *dynamic windowing*. With this approach, Alice's episode duration is dynamically updated as follows:

$$T_{\max,A}[t+1] = \begin{cases} \mathrm{T}_{\max,A}[t] + 1 \text{ , if Bob reached goal} \\ \max(\mathrm{T}_{\max,A}[t] - 1, 1), \text{ else} \end{cases} \,. \tag{6.2}$$

Although this addition might indeed keep Alice from proposing goals that are too hard for Bob and hence solve the A>B issue, sparse rewards are still less informative than more dense rewards which would seem to reduce Alice's power w.r.t. to the next reward formulation. Furthermore, this reward structure requires Alice to always execute episodes of the maximal duration, which can increase the training time of the framework significantly.

Time-based Rewards

This is the original formulation as proposed by Sukbaatar et al. [1], where Alice's episode reward is defined as follows:

$$r_A = \max(\gamma(t_B - t_A), 0) \,. \tag{6.3}$$

No direct motivation was given for the lower bound of zero on the reward. It would seem that removing this bound provides Alice with more information on Bob's performance, in particular in the case of B $>$ A, which would result in in a zero reward for all nearby goals (cf. [12]). Therefore the lower limit was removed and the resulting reward function reduces to:

$$r_A = \gamma(t_B - t_A) \,, \tag{6.4}$$

where γ is used to rescale the rewards to a smaller range, as the maximum episode duration is often well above 100. This thesis uses $\gamma = 0.01$ whenever the time-based reward is used.

In order to use this reward structure, Alice needs a way of choosing when to stop her current episode. Sukhbaatar and Al. accomplished this by adding an additional action to discrete action space that represented this STOP signal [1]. In continuous settings, however, this is not as straightforward.

An alternative is to create a separate stopping policy (either a completely separated network or a separate head of the policy network). An easier solution however could be to encode this STOP with the action norm: if the L2 norm divided by the action dimension is smaller than a certain constant, Alice's episode is stopped:

$$\text{STOP} = \frac{||a_i||_2}{\text{action\_dim}} < x\,. \tag{6.5}$$

Although this would prohibit Alice from taking very small actions, it seems like a low-overhead way to learn Alice how to signal STOP and evaluate this reward structure, which is expected to be more informative for Alice.

6.4 Bringing More Diversity to Bob's Goals

As discussed in the first section, the main expected issue is that because of Alice's unimodal policy, she would only present a small subset of the goal space at a time to Bob. In this section, a few potential methods for mitigating the resulting issues are discussed.

Multiple Alices

A first way of trying to deal with Alice's tendency to converge would be to add multiple Alice's which could be scheduled (e.g. in a Round Robin fashion) for proposing a goal to Bob. This would present more diverse goals to Bob on the one hand and also reduce the probability of the Alices getting stuck. Sukbaatar and Al. also suggested this to increase diversity [1].

Such an approach has one immediate disadvantage: Since all Alices would be trained separately, their learning progress would be inversely proportional to the number of distinct Alices. This might pose difficulties for the Alices to move to different regions of the goal space or to even reach harder goals in the goal space.

Another possible issue would be that the Alices are not guaranteed to cover different parts of the goal space, since they are not aware of each other.

Conditioning on Random Initial States

For robotics, there usually is the notion of a "home position" for a manipulator, to which we can easily reset the controller. Therefore one is usually only interested in reaching all goals from this home position, which is why the initial pose can be fixed to reduce the task complexity. Often the initial state is still randomized as a way to increase explorational power for the agent (Bob). However, with asymmetric self-play this is not necessary as Alice is supposed to present Bob with appropriate goals.

Relating to Generative Adversarial Networks (GAN) [47], which share some similarities with the asymmetric self-play approach, it seems that a possible solution for the convergence issue might be to add some random prior to the generator, or in this case to Alice. Randomizing the initial state could serve this purpose and allow Alice to condition her actions and hence the goals she offers to Bob, on this random initial state. This would allow her to create a somewhat multimodal goal distribution after all.

This conditioning could be done either explicitly in the policy input: $\pi_A(a|s, s_0)$, or implicitly by using recurrent neural networks. OpenAI uses recurrent networks [11], although they do not explicitly mention this conditioning as motivation. Using recurrent neural networks as function approximators instead of simple multi-layer perceptrons has been found to increase learning speed and performance [28], so it is very well possible that this was the reason for introducing them. Explicit conditioning seems more straightforward and hence this option is used here.

Interleaving with Random Goals

A final way of increasing the diversity of the goals presented to Bob would be to interleave self-play episodes with random goal selection, inspired by original formulation where self-play was combined with regular play [1]. However, these goals would be most likely be too hard to reach for Bob when using binary rewards in a multi-goal setting, resulting in non-informative experiences.

6.5 Evaluating Asymmetric Self-Play on Low-Dimensional Task

In this section, the ASP implementation from Section 6.2 and some of the different reward structures and other modifications will be evaluated on the low-dimensional pointmass environment that was introduced in Chapter 4.

These experiments have three purposes:

1. First of all they serve as a test for the implementation of the asymmetric self-play framework.
2. Secondly, they serve to evaluate which of the discussed issues are showing up and how they limit the performance on this low-dimensional task.
3. Finally the experiments allow for evaluating the influence of the different reward structures and other modifications.

The performance metric is the average success rate over 20 evaluation episodes, for which the goals are sampled uniformly from the goal space.

To make the results more robust and limit the influence of random seeds as suggested in [36], all experiments are repeated with 5 different seeds where both the mean and standard deviation are reported for the performance metric.

6.5.1 Comparing Reward Structures

In this experiment, sparse rewards for Alice are compared to time-based rewards using a 1D-pointmass. The hyperparameters that are used for Alice and Bob are the default parameters from Chapter 5 with some exceptions:

- Alice's rollout length is now set to her episode duration and only a single batch is used during training.
- Alice's learning rate is set to 0.001.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
learning rate	0.001	actor learning rate	0.0001
ppo clip value ϵ	0.2	noise std deviation	0.01
gradient clip norm	0.5	critic lr	0.001
batch size	previous episode duration	batch size	128
epochs	10	train frequency	1/step
rollout length	previous episode duration	buffer size	1e6
discount factor	0.99	discount factor	0.99
GAE factor	0.95	policy update frequency	2
value loss coefficient	0.5	target noise parameter	0.2
entropy loss coefficient	0.0	target noise clip value	0.5
max episode duration	100	max episode duration	100

Table 6.1 Default hyperparameters for Alice and Bob on the pointmass environment

- Bob's exploration noise standard deviation is set to 0.01 to artificially increase the task difficulty for Bob and to make Alice work harder.

Both Alice and Bob have a maximum episode duration of 100, which should be more than enough to reach the other side of the state space. The initial state of each episode is fixed to the origin. All hyperparameters are listed in Table 6.1.

The sparse reward structure is evaluated with and without the *dynamic windowing* addition. For the time-based rewards, 0.2 is used as the fraction for the norm threshold. Whenever the L2 norm of Alice's actions is below this value, this is interpreted as the STOP signal. The experiment was run for 400 episodes. Bob's performance under the different reward shapes for Alice can be observed in Fig. 6.2, where the performance is evaluated every 50 episodes.

From this figure, it can be seen that the time-based reward structure clearly outperforms the two other methods, as expected. To see more clearly why this is happening, Fig. 6.3 shows the emerging curricula using time-based and sparse rewards. Here the goal (a 1D position) proposed by Alice during each episode is shown. The goals are color-coded to indicate whether Bob was able to achieve them (green) or not (red). These curricula show how using time-based rewards results in a more dynamic curriculum, whereas the sparse rewards make Alice converge on one edge of the goal space resulting in her getting stuck once Bob has learned how to reach these goals. This confirms the initial hypothesis that using time-based rewards is more informative for Alice and will result in better performance of the framework.

Another observation from Fig. 6.2 is that *dynamic windowing* does not improve the results. This makes sense however as this addition was aimed at keeping Alice from proposing goals that are too difficult whereas the problem here is that Alice is actually gets stuck in a corner of the space and has no information on where to go once Bob starts to master this part of the goal space.

These findings are rather different from the ablation study performed by OpenAI on their asymmetric self-play framework in which they found the performance rather similar for both reward structures [11]. This can be most likely explained by the multimodal distribution Alice is

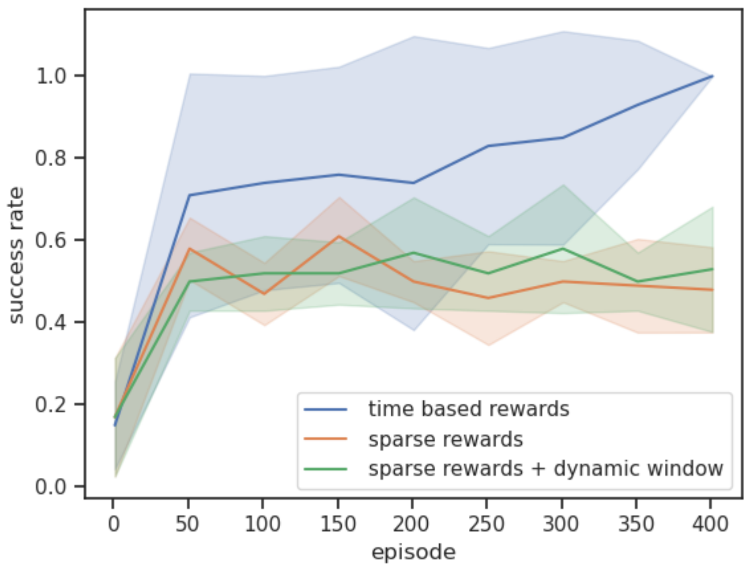
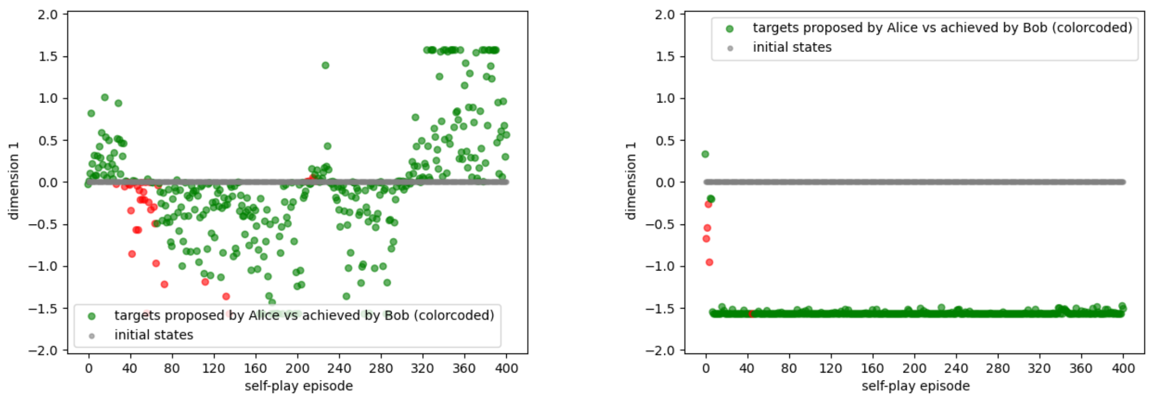


Fig. 6.2 Performance of Bob on the 1D pointmass environment using different reward structures for Alice.



(a) time-based rewards

(b) sparse rewards

Fig. 6.3 Comparison of the curricula for a single seed when using different reward structures for Alice. The color of each goal indicates whether Bob was able to reach this goal.

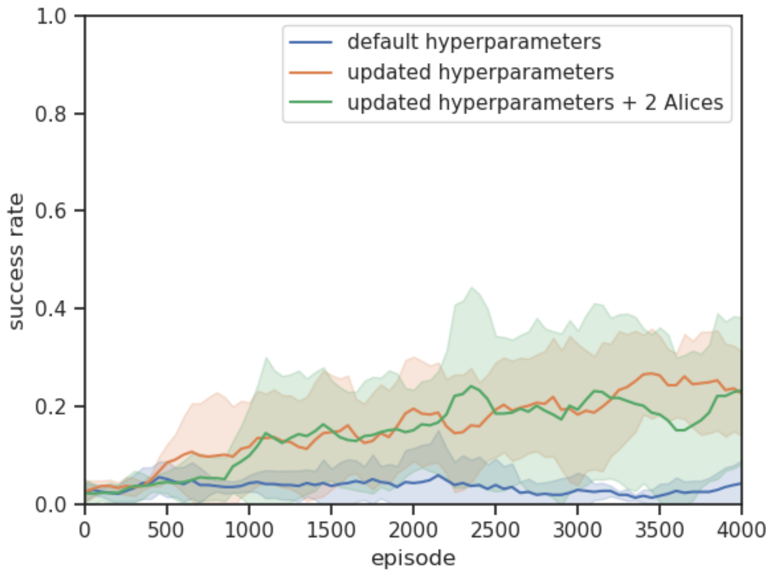


Fig. 6.4 Performance of Bob on the 2D point mass environment using fixed initial states with different configurations and hyperparameters for asymmetric self-play.

learning in their work, which would prevent her from getting completely stuck as there will always be a small probability of reaching a different part of the goal space. Furthermore, as mentioned before, they used behavioral cloning, which would limit the influence of Alice presenting goals that are too hard for Bob.

6.5.2 Evaluating Convergence

From the previous subsection, it has become clear that time-based rewards are indeed the best option. In this subsection, using this time-based reward structure, the asymmetric self-play framework will be evaluated on a 2D-pointmass environment. The goal of this series of experiments is to assess to which extent Alice exhibits converging behavior, how this limits Bob's performance, and whether the proposed countermeasures are able to reduce the impact.

The experiments continue for 4000 episodes where the initial state is always the origin unless stated differently. Evaluation takes place every 100 episodes using 20 random goals. As before the average success rate is reported for each evaluation as the performance measure. The results are smoothed using a moving average with a window of size 5.

The first experiment uses again the default hyperparameter values as reported in Table 6.1. As can be seen in Fig. 6.4, Bob's performance is very low. The resulting curriculum of proposed goals, which is shown in Fig. 6.5, shows clearly that Alice converges to a very small subspace and seems to indicate she might get stuck there somehow. However, taking a look at Bob's episode success rate during training (Fig. 6.5) reveals that the last 2000 goals proposed by Alice are actually never reached by Bob, in which case Alice has no incentive to move to different regions of the goal space.

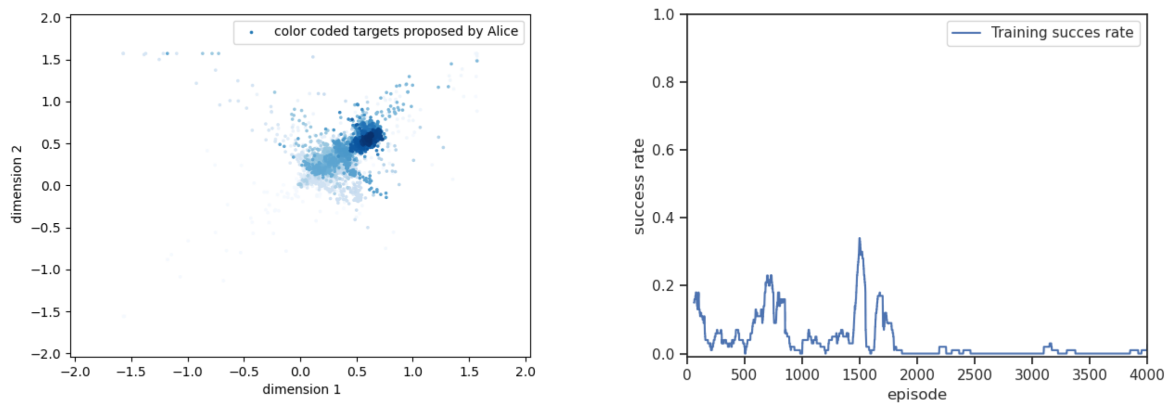


Fig. 6.5 Resulting Curriculum of goals proposed by Alice and Bob's training success rate on those goals, for the default hyperparameters. The goals in the curriculum are color coded to show the evolution over time, where darker goals are more recent.

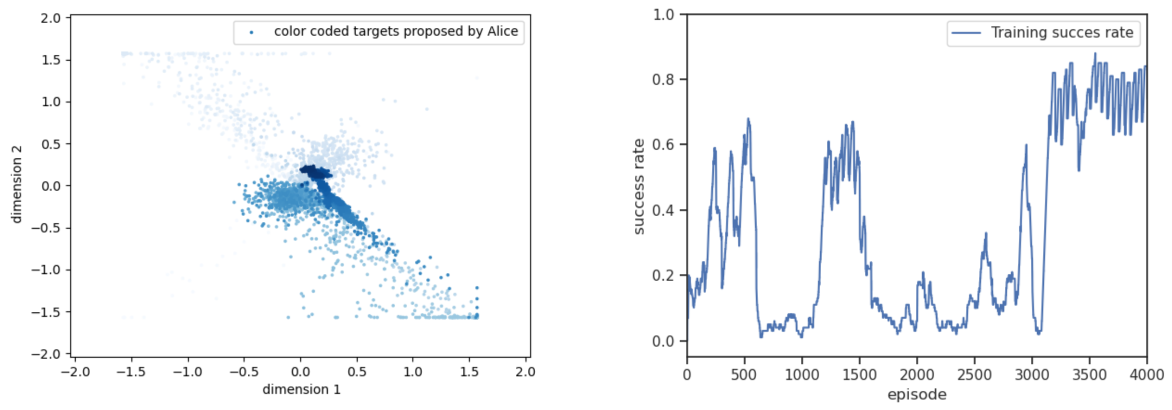


Fig. 6.6 Resulting Curriculum of goals proposed by Alice and Bob's training success rate on those goals, for the updated hyperparameters. The goals in the curriculum are color coded to show the evolution over time, where darker goals are more recent.

Relating to the anticipated issues of Section 6.1, the hypothesis is that Bob overfits on earlier goals proposed by Alice and has no way of reaching the subspace of goals Alice is proposing later on.

A first attempt to deal with this issue was to increase the noise on Bob's actions to a standard deviation of 0.2, decrease the batch size to 64 and decrease the number of train steps per episode from once per step to 10 per episode. The idea is that these changes will increase Bob's explorational power and decrease its training rate, to reduce the overfitting impact. As can be seen from Fig. 6.6 the resulting curriculum and Bob's training success rate already look more interesting. It can be observed now that Bob is indeed learning to achieve the goals Alice is proposing and that Alice seems to find new goal regions once Bob starts to master the current ones, which Bob then slowly starts to master again. However, the resulting performance is still rather low, as can be seen in Fig. 6.4. This seems to be caused by Alice presenting only small regions of the goal space at a given time, resulting in very slow learning progress for Bob. Hence, bringing more diversity to the goals presented to Bob is what should be aimed for next.

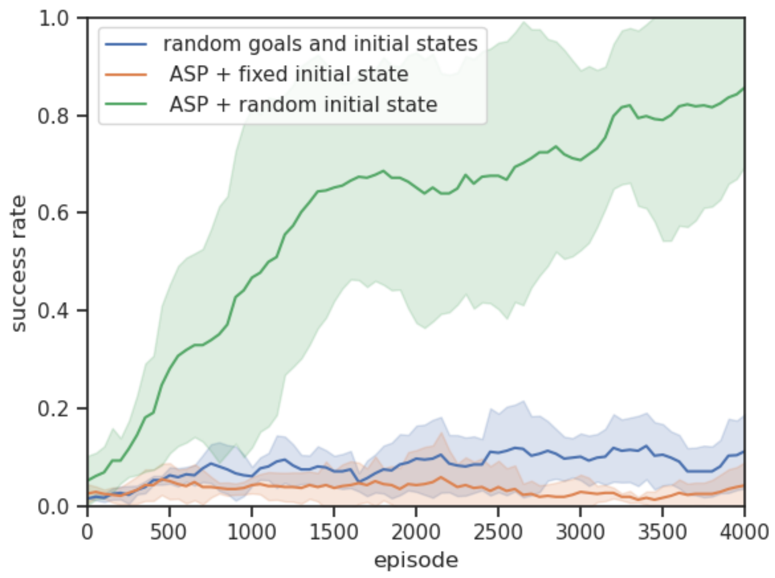


Fig. 6.7 Performance comparison of Bob on the 2D point mass environment using random initial states for asymmetric self-play.

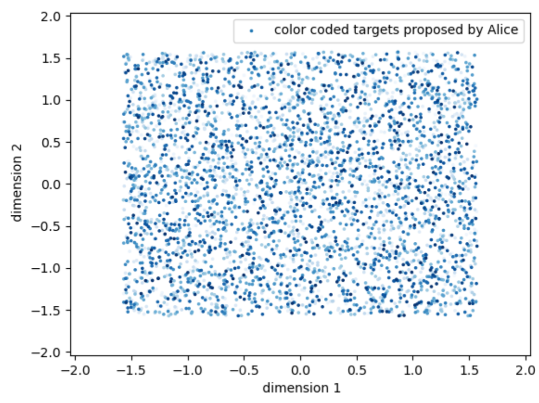
The first proposed solution was to use two distinct Alices and make them present a goal to Bob one after the other. However, as can be seen in Fig. 6.4, this actually did not increase performance. As both Alices are still only presenting Bob with a small part of the goal space, this does not solve the issue of Bob overfitting. Furthermore using multiple independent Alices sometimes resulted in them converging to neighboring goal regions.

In a second attempt to bring more diversity into Alice's goal selection, the initial state is now randomized over the state space. The hyperparameters are set back the original parameters from Table 6.1 to clearly evaluate the impact of using random initial states. The resulting performance can be seen in Fig. 6.7, where it is compared against random goal selection and ASP with fixed initial states. From this figure, it can be seen that asymmetric self-play with random initial states enables Bob to learn the task. Using random goals does not result in a good performance. From the same figure, it can also be seen that asymmetric self-play using the original hyperparameters and fixed initial states, actually performs worse than using random goals and initial states. This is most likely because Bob is getting stuck due to overfitting, as described before.

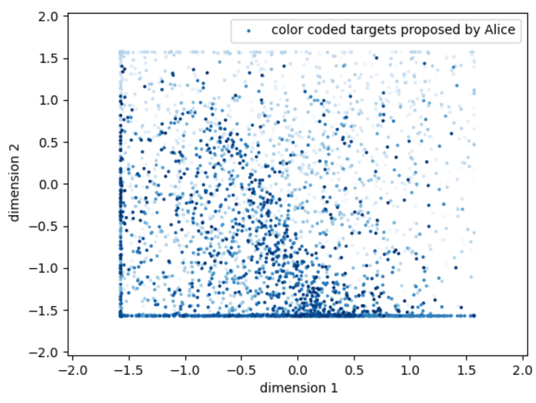
The resulting curriculum of a particular run is shown in Fig. 6.8. This curriculum shows largely improved coverage of the goal space in comparison to previous curricula. At the same time, it can be seen that Alice still starts to converge to certain regions of the goal space, although in a more multimodal way than before as she seems to explore multiple regions at the same time.

6.5.3 Evaluating Behavioral Cloning

In this last subsection, the impact of Behavioral Cloning (BC) on Bob's performance is evaluated. The hypothesis is that this addition could help Bob to escape from overfitting on Alice's earlier goals even if he does not manage to reach them himself, by explicitly adding a successful trajectory to the replay buffer.



(a) random goal selection



(b) asymmetric self-play

Fig. 6.8 Comparison of the curriculum resulting from asymmetric self-play using random initial states with the curriculum resulting from random goal selection. Goals are time-encoded where darker is more recent.

Using the same updated hyperparameters as proposed before to reduce Bob's overfitting issues, the performance with and without behavioral cloning is compared. As can be seen from Fig. 6.10, behavioral cloning increases Bob's performance, although with a significantly increased standard deviation.

To illustrate why this is the case, the resulting curricula for 2 different seeds are shown in Fig. 6.10. The first one shows a diverse curriculum that visits different regions of the goal space over time, indicating that Behavioral Cloning helps Bob to escape earlier overfitting, unlike in 6.6. This enables Alice to explore the goal space without being held back by Bob and as Bob learns to solve the goals proposed by Alice, this results in a good performance.

The second one, however, shows how Alice initially proposed goals from a very concentrated subspace and then moved on to a different part of the goal space. Since all these previous experiences are captured in the replay buffer, it makes sense that it takes quite some time for Bob to accumulate experiences using BC before he can undo the overfitting that has taken place. This explains the high variance. It is expected that if the learning were to be continued for another series of episodes, Bob would eventually be able to escape this local optimum with Behavioral Cloning.

At this point, it seems that using an off-policy algorithm for Bob could in some cases slow down learning since the experiences that are captured in the replay buffer might make it more difficult to undo the previous overfitting. By randomizing the initial states as discussed before, it seems that the goals provided by Alice are diverse enough to limit the impact of this non-balanced buffer content, while still benefiting from the mixing of all goals previously proposed by Alice in the replay buffer to reduce the impact of the convergence issues.

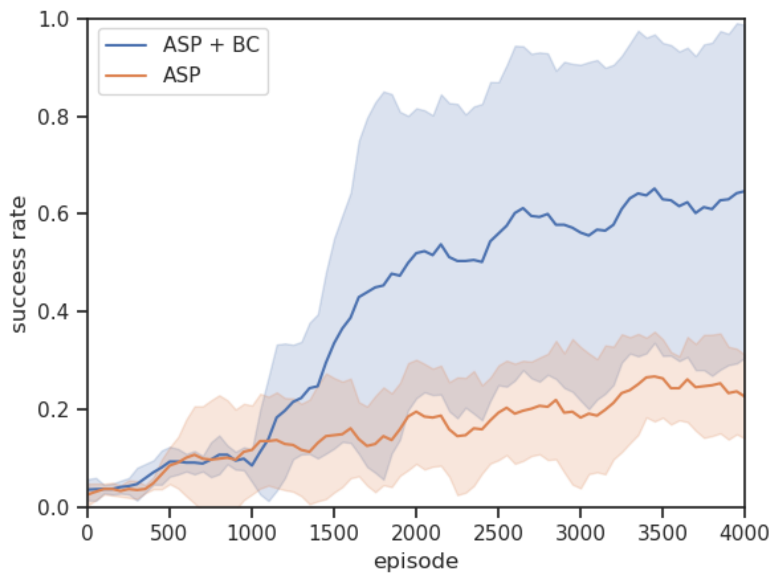
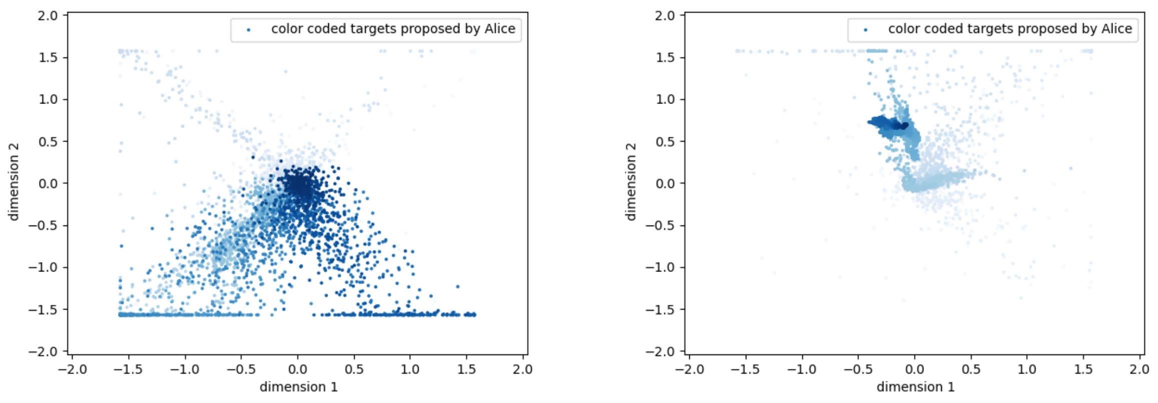


Fig. 6.9 Performance comparison of asymmetric self-play with and without Behavioral Cloning and fixed initial states using the updated hyperparameters.



(a) BC helps to escape overfitting

(b) BC unsuccessful in helping to escape overfitting

Fig. 6.10 Comparison of resulting curricula using Behavioral Cloning for different seeds. Goals are color-coded where darker means more recent.

6.6 Discussion

In this section, the main findings from the low-dimensional experiments are discussed.

The low-dimensional experiments first of all allowed to perform some *unittests* on the asymmetric self-play framework. Failure to learn the 1D-task with any reward structure, for example, would have been a strong indicator of bugs in the asymmetric self-play algorithm.

The experiments furthermore allow for making some conclusions on the use of asymmetric self-play (in continuous action spaces):

- Time-based rewards clearly outperform sparse rewards, which are prone to either Alice proposing goals that are too hard or Alice getting stuck once Bob has caught up on the goals she has converged to.
- Using the norm of the actions to encode the STOP signal for Alice seems to work rather well for these environments.
- Alice's convergence issue is indeed a fundamental problem for continuous action spaces. By proposing only a small subset of the goal space at each time, Bob tends to overfit on this subset and this results in the ASP framework getting stuck. Even if Alice is proposing new goals that are of an appropriate difficulty, Bob has already converged on the previous goals and does not have enough explorational power to reach the goals Alice is proposing.
- Using multiple Alices does not solve this issue, since even then Bob is apparently overfitting on the combined subspaces. Furthermore, it seems that multiple Alices sometimes converge to the same region of the goal space, as they are not aware of each other.
- Randomizing the shared initial state of Alice and Bob on the other hand seems to provide enough diversity to avoid this overfitting issue by allowing Alice to condition her actions on this initial state. It can be seen however that Alice's tendency to converge is still present and this might limit the use of asymmetric self-play on more difficult tasks.
- Behavioral Cloning seems to be a useful addition. It can help Bob to learn faster by learning directly from Alice's demonstrations and could also help Bob escape from overfitted goal regions. This can be slowed down however by overconcentrated experiences in the replay buffer as a consequence of convergence. This shows that using a replay buffer might actually be problematic if the experiences collected are not diverse enough. Randomizing the initial state seems to provide a solution.

Finally note that the addition of random initial states, multiple Alices, etc. with the purpose of avoiding the convergence issue of Alice, all introduce additional randomness to the goals proposed to Bob. This implies that it becomes harder and harder to assess how much of the credits should be given to Alice and how much of the performance increase is simply because Bob is learning from random goals.

To conclude, from the experiments in this chapter it has become clear that the main issue of asymmetric self-play in continuous action spaces is the tendency of Alice to converge to a small part of the goal state, caused by the unimodal policy representation. This leads to Bob overfitting on this subspace which results in the learning process being slow and unstable or even getting completely stuck in a local optimum for Bob.

Chapter 7

Learning Motion Planning using Asymmetric Self-play

In this chapter, the asymmetric self-play framework will be used to learn a control policy for a robotic manipulator using the Motion Planning environment described in Section 4.1.3. The objective is to further evaluate the asymmetric self-play framework described in the previous chapter on an actual robotics task with higher dimensionality and without straightforward relation between actions and goals.

In the first section, the methodology for the experiments is elaborated upon. The second and third sections contain the actual experiments, in which the task is first learned using dense rewards and then using binary rewards. Asymmetric self-play is also compared against hindsight experience replay. The fourth section contains a discussion on these results, where they are related to the results of Chapter 6 and of previous work on asymmetric self-play. This section also formulates some conclusions on the issues encountered with the asymmetric self-play framework, some future research directions to deal with these issues, and some conclusions on the pros and cons of both curriculum methods that were used. The final section again briefly discusses some of the lessons learned while performing the experiments with asymmetric self-play.

7.1 Methodology

As a performance metric for the motion planning task, 20 random goals are sampled from the goal space every 100 episodes, and the average success rate on these goals is reported. This metric does not explicitly capture information on the time the robot requires to reach the target position. Even though this would be of great importance for real-world industry applications it is not considered in this work to simplify the analysis and focus on whether the agent can reach the goals at all. As the discount factor is smaller than 1, the agent is still encouraged to reach the goals as fast as possible.

As has become clear from the previous chapter, randomizing the initial state is important to reduce the issues related to the convergence of Alice. Hence, this randomization will be applied to all experiments in this chapter.

Furthermore, it has become clear that certain hyperparameters have a large impact on the system. Hence for each experiment, a random search for some of the hyperparameters is performed with approximately 30 runs. To limit the computational resources required for the hyperparameter search, other parameters are set to default values for their respective learning algorithms. The hyperparameters of the run with the most promising evaluation score are then used to repeat the experiment with 5 different seeds, to make the resulting metrics more robust as it is known that these random seeds can have a large influence [36]. For each experiment, the average and standard deviation are reported.

The hyperparameter search contains the actor learning rate, number of epochs, batch size and noise standard deviation for Bob as these parameters have been found to influence the tendency of Bob to overfit on Alice's goals. The critic learning rate is kept at its default value since this did not seem to influence the outcome significantly on the pointmass environments. The size of Bob's hidden layers is also included, as some initial tests showed that this could have some impact.

For Alice, the entropy loss coefficient is included in the search, as this was expected to play a role in her tendency to converge. Other parameters for Alice are not included in the search, as the PPO algorithm is known for its stability and to reduce the computational requirements for each search.

Finally, the maximal episode durations for Alice and Bob are included in the search since these influence the explorational capacity of both agents.

All hyperparameters are listed in Table 7.1, where the search range is given for those that are included in the hyperparameter search. The resulting hyperparameter values for each experiment are included in Appendix B.

By accident, for some experiments the range for the entropy loss coefficient was set to $[0 - 0.005]$ instead of $[0 - 0.001]$, which was the intended search range. This is indicated in the appendix with an asterisk for completeness.

7.2 Learning Motion Planning with Dense Environment Rewards

In this section, the task is learned using a dense reward consisting of the Euclidean distance between the target position and the current end-effector position:

$$r(\mathbf{s}, \mathbf{a}, \mathbf{s}'|\mathbf{g}) = ||\mathbf{x}_{s'} - \mathbf{g}||_2 \,. \tag{7.1}$$

The purpose of these experiments is to explore Alice's capacity for goal discovery and to make a first evaluation of the behavior in higher dimensions and the influence of convergence on Bob's performance. Furthermore, as an additional check, it serves to evaluate whether this distance-based reward causes local minima for this task, a phenomenon which was discussed in Chapter 2.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
learning rate	0.001	actor learning rate	[5e-5,1e-3]
ppo clip value	0.2	noise std deviation	[0.1-0.4]
gradient clip norm	0.5	critic learning rate	0.001
batch size	previous episode duration	batch size	[32,64,128]
epochs	10	train frequency	[1-100]/episode
rollout length	previous episode duration	buffer size	1e6
discount factor	0.99	discount factor	0.99
GAE factor	0.95	policy update frequency	2
value loss coefficient	0.5	target noise parameter	0.2
entropy loss coefficient	[0.0-0.001 / 0.005*]	target noise clip value	0.5
hidden layer size	128	hidden layer size	[128,256,512]
max episode duration	[100,150,200]	max episode duration	[150,200]

Table 7.1 Hyperparameters for Alice and Bob on the motion planning environment. For hyperparameters that are included in the hyperparameter search, the range of possible values is given.

7.2.1 Learning with Random Goal Selection

For this experiment, we do not make Alice propose goals but simply use random goals sampled uniformly across the evaluation domain. Combined with the informative reward, this should allow Bob to learn the task.

As can be seen in Fig. 7.1 and from the learned behavior[1], Bob learns to control the manipulator and manages to reach almost all goals.

The conclusion from this experiment is that there are apparently no local optima for this dense reward function that limit Bob's performance. Furthermore, it shows that any failure to learn in later experiments is not caused by issues with the learning environment or evaluation method.

7.2.2 Learning with Asymmetric Self-Play

For the next experiments, the random goal selection is replaced by the asymmetric self-play framework.

As can be seen from Fig. 7.1, the resulting performance is lower than for the random goal selection. By adding behavioral cloning to the framework, the performance increases slightly and shows less variance, but still does not reach the level achieved by random goal selection.

Clearly, there must be goals that Bob still does not know to reach, yet there is no increase in performance anymore. Relating to the issues discussed in Section 6.1, there seem to be two possible explanations. The first being that Alice is stuck and cannot find these goals anymore. The second that Alice is still presenting Bob with challenging goals but that he is overfitting on the goals presented by Alice and hence does not improve on the evaluation success rate.

[1]https://youtu.be/CLtr6k61CaU

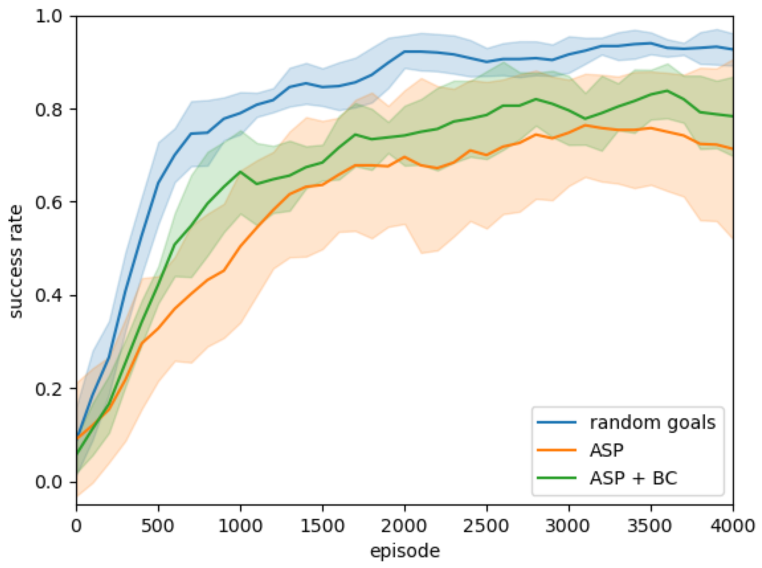


Fig. 7.1 Performance comparison of asymmetric self-play (ASP), asymmetric self-play with behavioral cloning (ASP + BC) and random goal selection using dense rewards.

Looking at the training success rate from Fig. 7.2, it is clear that Bob manages to solve most of the goals presented by Alice, indicating that he has not overfitted completely on earlier goals. At the same time, the wave-shape of the success rate also seems to indicate that Alice is still evolving and finding goals that Bob does not yet know how to solve. This is also confirmed by looking at the curriculum of goals that Alice is proposing, as can be seen in Fig. 7.3.

This seems to imply that even though Alice is still finding relevant goals for Bob and Bob is still learning to solve them, Bob is making no progress on the evaluation tasks. A possible explanation could be that Bob is in fact still overfitting slightly on more recent goals, and hence tends to forget how to reach certain goals he was previously able to reach even though these goals would still be in the replay buffer. If Alice converges really hard on some parts of the goal space, this would fill the buffer more than proportional with goals from this region of the goal space, making Bob overfit slightly on certain parts of the goal space.

Looking again at the curriculum of a particular run as showed in Fig. 7.3 seems to support this hypothesis. Initially, Alice is proposing rather diverse goals, but later on, she starts to focus on certain parts of the goal space that are hard for Bob at that moment. This results in very concentrated goals which indeed could make Bob overfit and help explain the stagnation in Bob's performance although both Alice and Bob are doing what they were instructed: Alice is finding goals that are hard to reach and Bob is learning to solve them.

Another observation from Fig. 7.1 is that the initial learning speed did actually reduce w.r.t. random goal selection. This suggests that learning benefits from enough diversity in the goal selection (if the experiences are informative, which is indeed the case using these dense rewards).

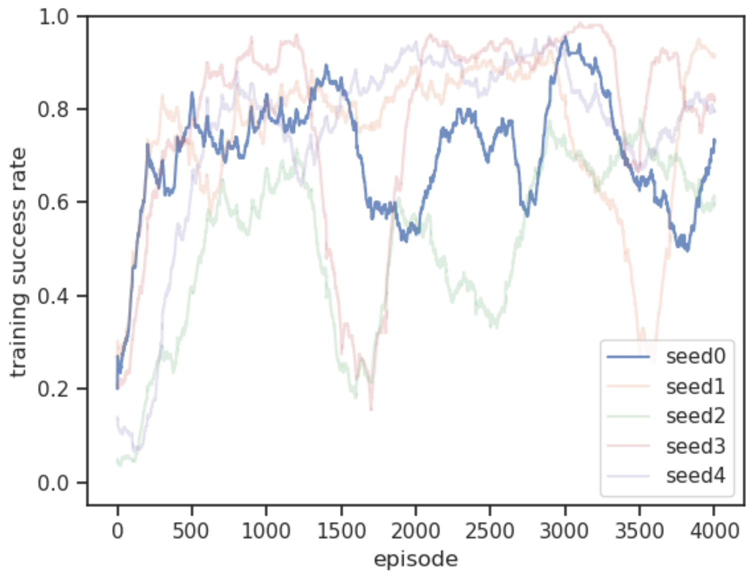
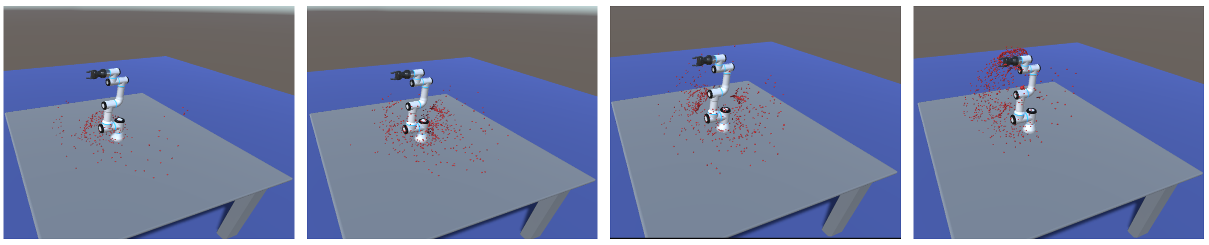


Fig. 7.2 Training success rates for Bob when training with asymmetric self-play (ASP) on dense rewards.



(a) 0 - 1000 (b) 1000 - 2000 (c) 2000 - 3000 (d) 3000 - 4000

Fig. 7.3 Curriculum created by Alice for Bob using dense rewards and asymmetric self-play. Each frame contains the goals proposed by Alice during 1000 consecutive episodes.

Finally, note that Alice found goals that are outside of the quarter sphere used for evaluation but are still reachable by Bob. This can for example be seen in Fig. 7.3 (d), where most of the goals are behind the robot although the quarter sphere only covers the space in front of the robot. This confirms that ASP could indeed be used for goal-discovery in situations where describing the (entire) goal space mathematically in order to sample from it is not trivial, and is related to the findings in [11]. However, whether this would also improve the generalization to goals within the target goal space cannot be assessed as the algorithm gets stuck due to overfitting.

7.3 Learning Motion Planning with Sparse Environment Rewards

In this section, the task is learned using binary rewards, 1 if the goal is reached (again using a threshold of 5 cm) and zero otherwise:

$$r(\mathbf{s}, \mathbf{a}, \mathbf{s}'|\mathbf{g}) = \begin{cases} 1, & \text{if } ||\mathbf{x}_{s'} - \mathbf{g}||_2 < 0.05 \\ 0, & \text{else} \end{cases} . \tag{7.2}$$

In the previous section, Alice did not have to care too much about presenting goals that were not too hard and could focus on discovery and exploration since with dense rewards every transition is informative for Bob. In this section, however, Alice's task is more difficult as Bob now requires goals that are too easy nor too hard, to make the resulting experiences contain sufficient informative rewards.

7.3.1 Learning with Random Goal Selection

When presenting Bob with random goals, the agent is not able to learn anything, as can be seen in Fig. 7.4. This should not be a surprise as the probability of Bob reaching a random goal is very small and hence it will almost never collect informative experiences, resulting in no learning progress.

7.3.2 Learning with Asymmetric Self-Play

For these experiments, the random goal selection is again replaced by asymmetric self-play. A number of different additions and combinations are evaluated and their performance is analyzed.

Asymmetric Self-Play

From Fig. 7.4, it is immediately clear that ASP improves on random goal selection but does not result in a good performance. As before, looking at Bob's training success rate in Fig. 7.5 shows that Bob manages to solve a fraction of the goals presented by Alice and again the drops in the success rate show that Alice is finding new parts of the goal region. However, when further comparing the training success rate to the success rate obtained using dense rewards, it is significantly lower. This is most likely due to the increased explorational difficulty for Bob. Previously whenever Alice proposed a goal, every step Bob took was informative as the reward

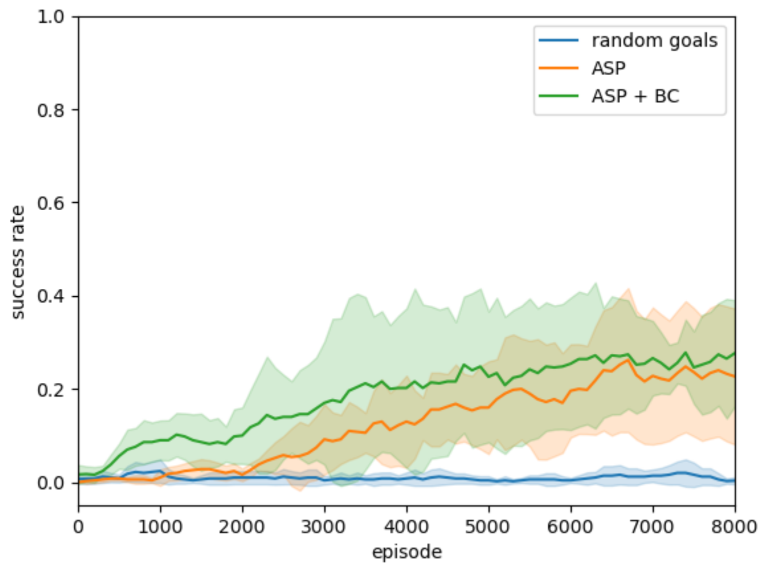
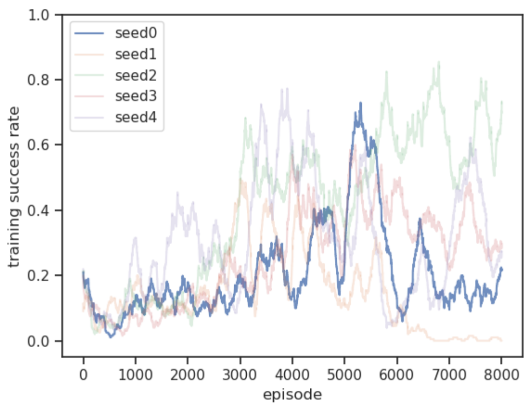
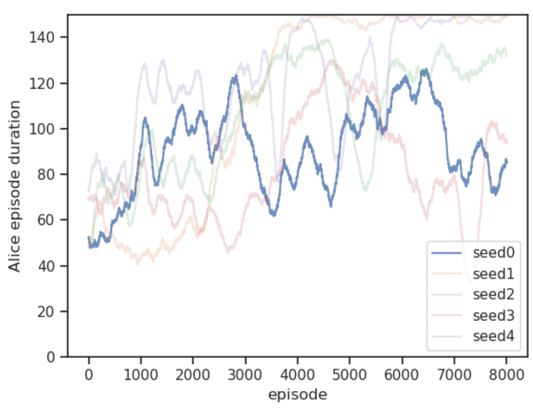


Fig. 7.4 Performance comparison of asymmetric self-play (ASP), asymmetric self-play with behavioral cloning (ASP + BC) and random goal selection using sparse rewards.



(a) Training success rate for Bob



(b) Episode duration for Alice

Fig. 7.5 Training metrics for asymmetric self-play on sparse rewards.

always provided guidance on how to change its behavior. With sparse rewards, however, only when Bob actually reaches the goal, the episode becomes informative, making it harder for Bob to learn new goals presented by Alice.

Furthermore, Alice's episode durations, which are also shown in Fig. 7.5, indicate that although she knows how to stop, she does not present Bob with nearby[2] goals. This might further explain the low training success rate.

Asymmetric Self-Play + Behavioral Cloning

Adding Behavioral Cloning did not improve the asymptotic performance much as can be seen in Fig. 7.4. This was unexpected as it did improve results before with dense rewards and in Chapter 6. Furthermore, the resulting behavior of Bob is often very shaky[3]. This seems to be caused by the best runs of the hyperparameter search having a high entropy loss coefficient for Alice. This results in a high standard deviation and hence jerky motion for Alice, and because of the behavioral cloning this jerky motion is transferred to Bob. This would prohibit any transfer of this policy to a real robot and highlights that BC on Alice's trajectories should be used with care since she is by no means an 'expert', as is usually the case for the demonstrations used in BC.

Manually lowering the entropy loss coefficient avoided this shaky behavior but resulted in very poor performance. So although BC increases the initial learning rate slightly, it does not improve asymptotic performance.

Asymmetric Self-Play + Dynamic Windowing

In an attempt to force Alice to reduce her episode duration and to increase the diversity of the goals proposed to Bob, *dynamic windowing* was added to the asymmetric self-play. This mechanism was already introduced in Section 6.3. In an additional attempt to avoid overfitting on the goals proposed by Alice, next to the *dynamic windowing* to set the maximal episode duration for Alice, random goals are presented to Bob every 4 episodes. This ratio was found to be the optimal choice with an initial search.

From the resulting performance in Fig. 7.6, it is clear that adding this restriction on Alice's episode duration further increases performance, yet again it seems to plateau eventually. Fig. 7.7 shows the resulting curriculum of a particular run. Initially, the goals proposed by Alice are diverse as expected due to the restricted episode durations. Later on, however, Alice starts to converge again and concentrates only on a particular part of the goal space instead of presenting Bob with more diverse goals. This most likely results again in overfitting of Bob, explaining why the learning progress is stagnating again. A video of the training process[4] for snapshots of Alice and Bob, confirms that Alice has converged again and is only proposing goals from the region of the goal space corresponding to Fig. 7.7 (d).

[2]Note that the MDP-distance is used as the distance metric, which makes the episode duration a good indicator for the distance.

[3]https://youtu.be/eTXdMEOeqyQ

[4]https://youtu.be/H5aOgx-szhE

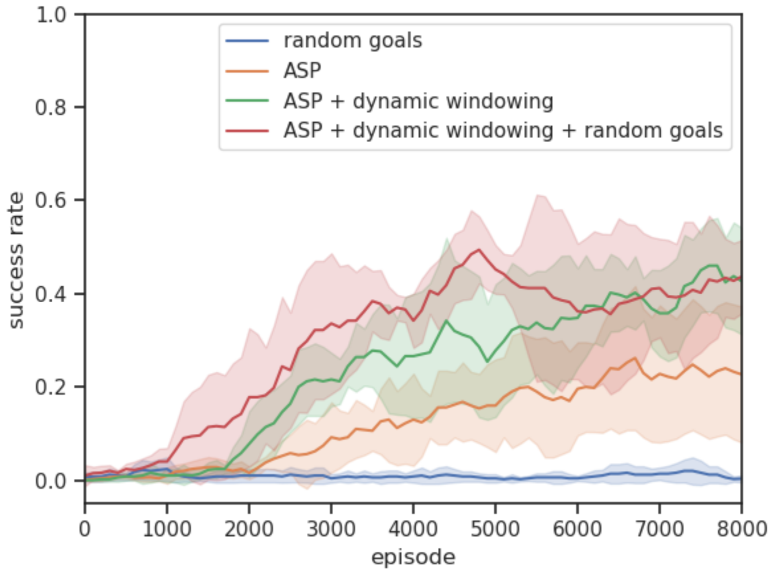
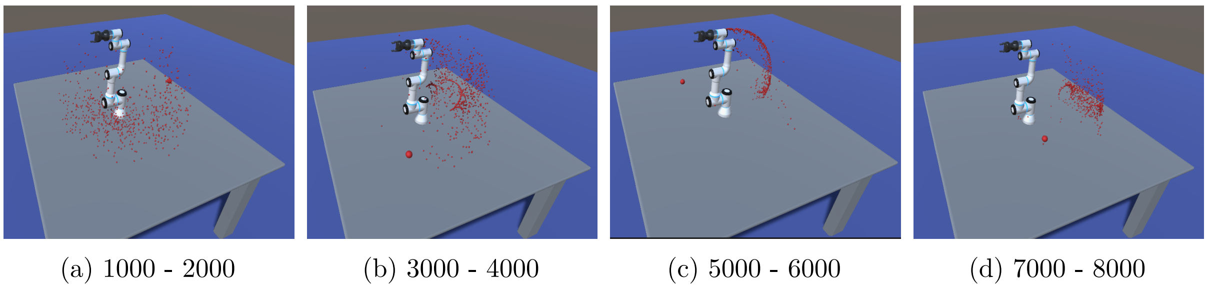


Fig. 7.6 Performance comparison of asymmetric self-play (ASP) using dynamic windowing and random goals on sparse rewards.



(a) 1000 - 2000 (b) 3000 - 4000 (c) 5000 - 6000 (d) 7000 - 8000

Fig. 7.7 Curriculum created by Alice for Bob using sparse rewards and asymmetric self-play with dynamic windowing to restrict Alice's episode duration. Each frame contains the goals proposed by Alice during 1000 consecutive episodes.

Adding random goals apparently did not help to avoid this, as Bob plateaus at the same success rate. This is most likely because the majority of these goals are not reachable, which again makes their episodes not informative. This is in contrast with the original work on asymmetric self-play [1], where rewards were dense and hence all trajectories were somewhat informative.

By restricting Alice's episode duration additional randomness was introduced again to the goals proposed by Alice, and this increased the performance. At this point, it seems that asymmetric self-play, because of its convergence, is simply not well suited for this task and that a method that provides more diverse goals of an appropriate difficulty should give better results. In the next section, this is explored further.

7.3.3 Learning with Hindsight Experience Replay

From the previous subsection, it has become clear that learning this task using sparse rewards and the asymmetric self-play framework does not result in optimal performance. In this section,

Hyperparameter	**Value**
learning algorithm	TD3
learning rates	0.001
noise std deviation	0.4
train frequency	50 / episode
batch size	64
replay strategy	`future`
replay goals / transition	4

Table 7.2 Hyperparameters for Hindsight Experience Replay.

hindsight experience replay (HER) will be used to create a goal curriculum for Bob, who still uses the TD3 algorithm for learning.

HER was introduced in Chapter 2 and has been shown to perform well on similar robotics tasks [16]. This experiment uses the implementation of the Stable Baselines framework [34]. The replay heuristic that was used is the one referred to as `future` in the original paper. With this replay method, for each transition, the goals achieved in k random future transitions from the same episode trajectory are used as the artificial target goals for the current transition. The number of replays was set to 4, which was shown to be a good choice in [16]. Other hyperparameters were simply chosen similar to the ASP hyperparameters or set to their default values. All hyperparameters are listed in Table 7.2.

Since the goals for HER are sampled randomly at training time, the performance metric used is simply the training success rate for HER, whereas for asymmetric self-play the same explicit evaluation is used as before. As usual, the experiment is repeated using 5 random seeds and the mean and standard deviation are reported.

The performance is compared against ASP and plotted against Bob's environment steps. This is different from previous figures, where the performance was always plotted against the episodes for the ASP framework. This is because Alice proposes a goal for each episode and hence it seemed more interesting to use the number of episodes as the independent variable. For this comparison, the number of environment steps is used, which is the default within the research community as it shows a direct relationship with the sample efficiency. Note that only the environment steps for Bob are taken into account. Alice's steps further decrease the total sample efficiency of the framework.

Furthermore, for HER, the performance is evaluated using random initial states whereas for ASP the initial state is always the home position. This is simply because the Stable Baselines implementation uses random initial environment states by default. Manual evaluation of the final policy using the fixed initial state however confirmed that this does not influence the asymptotic performance measure.

The resulting performance can be observed in Fig. 7.8. From this figure, it is very clear that HER outperforms ASP and is able to guide Bob to learn this task effectively using binary rewards. The success rate seems to stagnate around 0.8. This might be solved by searching for better

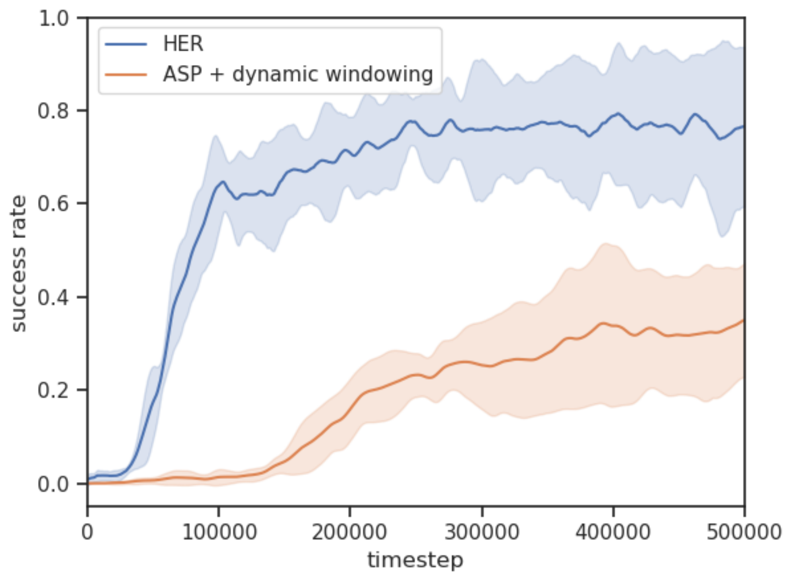


Fig. 7.8 Performance comparison of HER and ASP + dynamic windowing with sparse rewards.

hyperparameters. Visual observation of the resulting policy[5] however, seems to indicate that the goals not yet mastered by Bob are on the edges of the goal space. These goals are indeed expected to be harder to solve as there is usually only one joint configuration that can reach them. This already hints at a potential limitation of uniform sampling of goals, as used in HER.

7.4 Discussion

From the previous experiments, some conclusions can again be made w.r.t. learning motion planning and the use of asymmetric self-play:

- When using a distance-based dense reward, random goal selection allows Bob to reach almost perfect performance. This shows that local optima are not a severe problem for this particular task.
- When using sparse rewards, random goal selection no longer works as expected since the delayed reward makes almost all experiences non-informative. This shows how well-shaped dense rewards can improve learning significantly.
- Using asymmetric self-play with dense rewards results in suboptimal performance, caused by the tendency of Alice to converge on certain parts of the goal space on which Bob then overfits.
- With sparse rewards, asymmetric self-play improves on random goal selection but gets stuck on a low performance. This is again caused by the overfitting of Bob on certain parts of the goal space due to the convergence of Alice. Contrary to the situation with dense rewards, Bob has a very low training success rate. This is caused by the sparse rewards that make non-successful episodes not informative.

[5]https://youtu.be/4yORLbOMZSs

- The fact that Alice does not limit her episode duration more is unexpected and might be an additional reason for the limited performance when using sparse rewards. This might be caused by the norm-based STOP encoding and hence using a separate head should be tried. Next to this, tuning the reward function for Alice by adding more weight to her episode durations might be required.
- Behavioral Cloning can lead to very shaky policies. Furthermore, it does not seem to improve asymptotic performance with sparse rewards, which is unexpected.
- Restricting Alice's episode duration more explicitly using dynamic windowing results initially in more diverse goals and hence in higher performance. Later on, Alice again starts to converge and Bob overfits on these goals, resulting again in the system getting stuck.
- The issue of Alice getting stuck and not finding new goals to present to Bob, which was the main expected issue in [12], was not observed in this or the previous chapter. This might be related to the removal of the lower bound in Alice's reward function or the introduction of the replay buffer which partially prohibits overfitting. It could also be that the author misinterpreted the stagnation of the system as an issue of Alice getting stuck whereas it was also caused by Bob's overfitting.
- Hindsight Experience Replay was able to guide Bob to learn the task much faster and does not have the increased sample inefficiency that comes with Asymmetric self-play as there is no Alice that needs to interact with the environment.

It is clear from these findings and those of the previous chapter that asymmetric self-play with continuous action spaces suffers greatly from the convergence issue. This often results in Bob overfitting on certain parts of the goal space, which slows the learning process down and even results in a complete stagnation.

Using an off-policy algorithm with a replay buffer to smoothen the goals presented by Alice, as was done in this work, does not completely solve this overfitting issue. The hypothesis is that as Alice fills the buffer with concentrated goals, sampling uniformly from this replay buffer does not entirely avoid overfitting by Bob as the experiences themselves become too concentrated. Randomizing the initial states seemed to provide enough diversity in the goals presented by Alice on the low-dimensional task pointmass environment. On the motion planning task of this chapter on the other hand using a uniformly sampled replay buffer has proven not to be sufficient to overcome the overfitting that is caused by Alice's convergence. This is only a hypothesis however. Validating the impact of unbalanced data in the replay buffer requires using one of the suggestions in Section 7.4.1 to test this hypothesis.

Furthermore, replay buffers might make it harder for Alice to teach Bob new goals once Bob has overfitted as the probability for new experiences to be used in the next Q function update decreases as the buffer size increases.

These observations should draw explicit attention to the importance of having balanced data when using replay buffers. Using strongly non-uniform, time-varying distributions of goals and/or other elements in the MDP, which is often done in active curriculum learning methods, does not seem to go well with replay buffers because of the aforementioned reasons.

7.4.1 Future Work on Asymmetric Self-Play

Avoiding the Overfitting Issue

The most straightforward alternative to deal with the overfitting would be to continue using an on-policy algorithm for Bob and to somehow replay some previous goals of Alice to reduce the overfitting on the current goals Alice is proposing. The OpenAI paper on goal-discovery using asymmetric self-play [11] indeed mentions that both Alice and Bob play 20% of the time against past versions of their opponent to improve stability and avoid forgetting (which is induced by overfitting on more recent goals). This indicates that even in discrete action spaces the authors were confronted with overfitting issues. They however do not provide any details on how these past versions are sampled and did not include this past replaying in the accompanying pseudo-code, making it easy to miss this part of their work which is expected to be crucial. This approach also gives up the increased efficiency that comes with off-policy methods, which are usually an order of magnitude faster according to [6]. This sample efficiency becomes even more important when later on again combining goal curriculum learning with domain randomization, as was the initial idea based on [22], in which case Bob has to learn to achieve Alice's goals under different dynamics. Furthermore, it remains a question how well this solution would transfer to continuous actions spaces where the convergence issue is a lot more severe.

The notion that not every experience in the replay buffer might be equally informative at each moment in time was explored before by Schaul et al. in [48] to increase learning speed. They proposed Prioritized Experience Replay (PER), which uses a heuristic based on the error in the Q function as a measure for how informative an experience is and samples the next batch according to this measure. This could be an interesting approach to deal with the unbalanced data in the replay buffer without falling back to on-policy algorithms. Instead of using it to improve learning speed, this method could be used to overcome the overfitting that is caused by the unbalanced data in the buffer. Additionally, adding a bias towards the most recent experiences could help in discovering new parts of the goal space. It might however still be the case that the buffer simply does not contain enough variety, in which case overfitting would take place even with PER.

Behavioral cloning should be able to help overcome this stagnation while also increasing the learning speed. However two observations were made here: first of all, it can lead to bad behavior for Bob as Alice is not an expert and secondly, the improvements from BC were lower than expected on the motion planning task. A better way to integrate the trajectories from Alice might be to introduce a second replay buffer for Bob. Separating Bob's experiences from those of Alice enables for controlling the ratio of experiences for each batch in the Q function updates. It also enables to optimize the policy directly in a supervised way, using the regular imitation learning loss $(\pi(s) - a)^2$. This was also proposed in [49], where the authors explicitly worked on integrating behavioral cloning with off-policy RL methods. To anticipate the non-expert trajectories of Alice, their proposed Q-filter, with which the imitation loss is only used in case the Q value of Alice's action is higher than the Q value of the action proposed by Bob's policy, would also be a very interesting addition. This would make sure Bob can overcome suboptimal demonstrations by Alice.

Dealing with Convergence

Next to trying to deal with the overfitting issues caused by convergence, it would be even better to tackle the convergence issue itself as this is inherently slowing down the learning process even if it does not cause additional issues.

One approach could be to make Alice care not only about the performance of Bob on the current goals but also more directly about what we care about: Bob's generalization on the goal space. One way to try to achieve this could be to leverage the idea of curiosity-driven exploration [50], in which the prediction error made by an agent on the environment dynamics is used as an indicator for how well this state space region has been explored. Providing Alice with such information about Bob might give her increased motivation to propose more diverse goals instead of focusing on a single region of the goal space at a time or might simply push her away when she starts to converge, leading to less convergence and hence less overfitting for Bob.

A second approach could be to again make use of different Alices but to make them aware of each other. A possible method could be to use Stein Variational Policy Gradient (SVPG) [51], which creates N *particles* that are repulsed from each other based on the similarity of their network parameters yet use the experiences from other particles as well. Hence, these particles could tackle the two issues described with using multiple independent Alices (cf Chapter 6). SVPG was designed to increase explorational power by having the particles visit different parts of the state space. However, this makes it also interesting to avoid convergence issues with curriculum teachers. It has for example been used in an environment curriculum method [8], where it most likely served this same purpose of avoiding the environment aspects being too concentrated over time.

7.4.2 Hindsight Experience Replay

In previous paragraphs, some possible directions for future research on asymmetric self-play have been proposed to limit the impact of convergence and the resulting overfitting that was observed. However, from the experiments in Section 7.3.3 it has become clear that Hindsight Experience Replay actually provides a method that is easier to implement and monitor, sample-efficient as off-policy algorithms can be used, and less prone to overfitting issues as the goals are still randomly sampled.

It can therefore be concluded that even if one managed to solve the convergence issue or avoid the resulting tendency of Bob to overfit, Hindsight Experience Replay should be used in a first attempt to introduce curricula for exploration and generalization on environments where the desired goal-space can be described mathematically. A limitation of this method is probably that goal spaces should not contain harder subspaces for which the uniform sampling would most likely not suffice to learn them.

The performance of HER on problems where the goal space contains harder subspaces that are of interest seems not to have been evaluated before and this is yet another interesting research suggestion, as it would be relevant to e.g. evaluate the use of HER for picking items from boxes etc. It might very well be that using slightly non-uniform sampling strategies would allow for focusing on certain parts that are harder while still avoiding overfitting.

7.5 Lessons Learned

In this final section, some of the lessons learned while performing the experiments with asymmetric self-play from the last two chapters are briefly discussed.

The Influence of Dimensionality

The results from the experiments with ASP on the motion planning environment resulted in findings that do not always correspond with the conclusions obtained on the low dimensional pointmass environment. This again highlights the importance of selecting appropriate benchmark and test environments. They cannot be overly complex as then training time starts to slow down iteration time, yet if they are not complex enough it appears that some issues could be masked or simply not manifest themselves, as was also observed during the implementation of the learning algorithms in Chapter 5.

Multi-Agent Systems

Throughout the last two chapters, it has proven to be rather difficult to analyze this two-player system as both influence each other. Finding causality in the observations is hence not straightforward. For example, the low training success rate for Bob was interpreted as a problem of Alice not proposing appropriate goals whereas it turned out that actually the problem was also caused by Bob. Visualizing the curricula and resulting behavior of the agents proved to be very helpful, as not everything can be expressed in training metrics.

Replay Buffer Issues

In hindsight, the issues with the replay buffer could have been foreseen and it seems unlikely that other researchers were not aware of this.

Upon looking back at different papers on curriculum learning, I indeed found that most of them use on-policy methods which seems to suggest that some researchers might have been aware of these issues. The paper on Active Domain Randomization [8] however uses a replay buffer with a teacher to set environment curricula, which made me wonder why they did not encounter this issue. Diving a little deeper in the appendices, the authors mention that they reset and randomly initialize their curriculum setting agents every 50 episodes, most likely to avoid the issues with overconcentrated data that were discussed in the previous section.

It is unfortunate that none of them documented this issue, as it seems a useful insight for other researchers working on curriculum learning. This is again a confirmation that scientific progress does not only come from publishing success stories and things that work, but also from providing insight into things that did not work or limitations of current approaches.

Reward Shaping

The results of Chapter 7 have also confirmed how much difference a well-shaped reward function can make. Exploration becomes tremendously difficult when using binary rewards and hence from a pragmatic point of view, it seems very much worth the trouble of experimenting with reward shaping before deciding to move to binary rewards to avoid local optima, if present. Creating a dense reward function without local optima did not exactly require expert knowledge for the motion planning task, but for more complex tasks it has been shown to be more time-consuming and to require expert knowledge on the target domain [52]. From a research perspective, binary rewards are still of much interest as they are completely domain agnostic.

Chapter 8

Conclusion

In this work, asymmetric self-play was explored as a method for generating goal curricula in environments with continuous action spaces. This research was performed bottom-up, meaning that the whole process of creating a complex DeepRL framework to solve custom problems was described and some lessons learned were discussed.

First, a simulation environment to learn various behaviors for an industrial robot using joint position control was developed. Using this environment, a motion planning task was created in which the robot has to bring the end-effector to a specified location.

For the teacher and student in the asymmetric self-play framework, two state-of-the-art model-free learning algorithms were implemented and tested. Using these learning algorithms, a goal-conditioned implementation of ASP was made. To anticipate on the expected convergence issues for Alice and to make the framework more sample efficient, a replay buffer was introduced for the student Bob.

Using a low-dimensional pointmass task, different reward mechanisms and other modifications were evaluated. From these experiments, it became clear that asymmetric self-play easily gets stuck due to the convergence of the teacher Alice and resulting overfitting of Bob on the goals proposed by Alice. Randomizing the initial states and using an informative reward function for Alice allowed to overcome this overfitting and to learn the pointmass task from binary rewards.

For the motion planning task it was shown that when using uniform goal selection, a distance-based, dense reward allows for learning this task without any curriculum. Learning from binary rewards, on the other hand, requires guidance to overcome the exploration issues. This confirms that although dense rewards often introduce local optima and hence require manual engineering, they are from a pragmatic perspective still very valuable.

When using dense rewards, the asymmetric self-play framework was found to reduce the performance w.r.t. random goal selection on the motion planning task. With sparse rewards, the framework outperforms random goal selection but although multiple modifications were tried, the performance plateaus at a suboptimal level. This is caused by the convergence issue which makes the student overfit on the teacher. The use of a replay buffer proved not sufficient to overcome this. The hypothesis is that replay buffers with uniform sampling inherently do not go well with highly concentrated and time-varying goal distributions as they become too unbalanced themselves.

Finally, hindsight experience replay was briefly explored as an alternative for asymmetric self-play. This method is easier to implement and analyze, while resulting in a better performance on the motion planning task. It requires however that the goal space can be sampled from and is limited by the uniform goal-sampling strategy.

To conclude, it is clear that asymmetric self-play, in theory, has a lot of potential for guiding learning w.r.t. exploration, generalization and goal-discovery. For now, its performance is mainly limited due to a lack of diversity in the goals presented by Alice. This results in a reduced learning speed and often even in complete stagnation as Bob overfits on these goals. It is expected that with further research these issues will be solved. However, the solutions will most likely make the framework even more complex, harder to analyze and less efficient.

Future Work

The first issue to overcome is the overfitting of Bob on Alice's goals. Using a replay buffer with uniform sampling did not solve this and hence the next step would be to use non-uniform sampling to deal with the unbalanced buffer content, which is expected to be the cause of the overfitting. Prioritized experience replay [48] provides a possible solution for this by sampling the experiences based on the TD-error. Additionally biasing the probability slightly towards the most recent experiences would likely help in learning new goal-regions faster.

To make better use of the trajectories provided by Alice, which is one of the main advantages of asymmetric self-play, a separate replay buffer should be used for the behavioral cloning trajectories. This allows to fix the ratio in each batch during the Q-function update. Additionally, it allows for using the supervised imitation loss to update the policy function directly. Both modifications were proposed in [49].

These updates should allow for overcoming the overfitting and hence should result in increased performance. In a next step, the convergence issue itself should be tackled as this inherently slows down learning. To this end, multiple Alices could be introduced and made aware of each other using Stein Variational Policy Gradient (SVPG) [51].

From a pragmatic point of view, however, reward shaping or more heuristic curriculum methods such as Hindsight Experience Replay seem to be more efficient for creating guidance, and hence these should be tried before turning towards asymmetric self-play.

Epilogue

In past years as well as during the course of this thesis, I have been amazed many times by the achievements of Deep Reinforcement Learning: learning fine-grained dexterous manipulation, learning to beat the world champion in the game of GO or learning end-to-end robotics control policies... At the same time, however, during this thesis, I learned that the proof of the pudding is in the eating.

The deepRL research field seems very much focused on success stories. Published papers often focus on how well their methods work and sometimes seem to hide or focus too little on negative results and/or limitations. An observed phenomenon for example is describing these limitations in appendices (and hence I now always start by looking at the appendices before reading a paper). Another is only describing final results and not indicating how hard it was to obtain them or how brittle component X or Y actually is, which is not captured in the mathematical formulations used to describe them. This focus on success results in my opinion in two problems:

1. New researchers (like myself) have unrealistic expectations of what can be achieved and overly optimistic views on the robustness of deepRL methods. This can be very discouraging as these researchers embark on overly ambitious projects in which they then risk getting stuck. Furthermore, it can be very unproductive as it does not emphasize enough how important details are and hence how carefully one should conduct research in RL, something which is then only discovered later on during the process.

2. Even more important: By focusing mainly on these success stories, knowledge is lost or not optimally spread to a broader research community, which slows down overall progress in the field.

DeepRL has achieved incredible things in recent years. It is an incredibly promising framework for learning optimal decision-making strategies in a domain-agnostic way. At the very same time, the field seems to suffer from a hunger for success. A more balanced focus on both the successes and limitations or issues, would create a more realistic view on the progress and state of the art. This would help in spreading knowledge which would result in better decision-making on which framework to use for a problem or make it easier to identify key bottlenecks in current methods. All of this would result in more overall progress, which is what we all strive for in the end.

References

[1] S. Sukhbaatar, Z. Lin, I. Kostrikov, G. Synnaeve, A. Szlam, and R. Fergus, "Intrinsic motivation and automatic curricula via asymmetric self-play," in *6th International Conference on Learning Representations (ICLR)*, 2018.

[2] International Federation of Robotics (IFR), "Industrial robotics." https://ifr.org/industrial-robots, 2020. Accessed 2020-05-07.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, p. 26–38, Nov 2017.

[4] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research (IJRR)*, vol. 37, no. 4-5, pp. 421–436, 2018.

[5] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[6] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train your robot with deep reinforcement learning: lessons we have learned," *The International Journal of Robotics Research (IJRR)*, vol. 40, no. 4-5, pp. 698–721, 2021.

[7] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 23–30, 2017.

[8] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization," in *Conference on Robot Learning (CoRL)*, pp. 1162–1176, PMLR, 2020.

[9] V. Kumar, D. Hoeller, B. Sundaralingam, J. Tremblay, and S. Birchfield, "Joint space control via deep reinforcement learning," *arXiv preprint arXiv:2011.06332*, 2020.

[10] R. Portelas, C. Colas, L. Weng, K. Hofmann, and P.-Y. Oudeyer, "Automatic curriculum learning for deep rl: A short survey," *arXiv preprint arXiv:2003.04664*, 2020.

[11] O. OpenAI, M. Plappert, R. Sampedro, T. Xu, I. Akkaya, V. Kosaraju, P. Welinder, R. D'Sa, A. Petron, H. P. d. O. Pinto, *et al.*, "Asymmetric self-play for automatic goal discovery in robotic manipulation," *arXiv preprint arXiv:2101.04882*, 2021.

[12] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel, "Reverse curriculum generation for reinforcement learning," in *Conference on robot learning (CoRL)*, pp. 482–495, PMLR, 2017.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[14] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[15] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *International conference on machine learning (ICML)*, pp. 1312–1320, PMLR, 2015.

[16] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Representation Learning (ICLR)*, 2015.

[19] O. G. Selfridge, R. S. Sutton, and A. G. Barto, "Training and tracking in robotics.," in *International Joint Conferences on Artificial Intelligence (IJCAI)*, pp. 670–672, 1985.

[20] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning (ICML)*, pp. 41–48, 2009.

[21] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey.," *Journal of Machine Learning Research (JMLR)*, vol. 10, no. 7, 2009.

[22] S. C. Raparthy, B. Mehta, F. Golemo, and L. Paull, "Generating automatic curricula via self-supervised active domain randomization," *arXiv preprint arXiv:2002.07911*, 2020.

[23] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, *et al.*, "Multi-goal reinforcement learning: Challenging robotics environments and request for research," *arXiv preprint arXiv:1802.09464*, 2018.

[24] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5026–5033, 2012.

[25] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." http://pybullet.org, 2016–2021. Accessed 2020-4-16.

[26] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2020.

[27] C. Florensa, Y. Duan, and P. Abbeel, "Stochastic neural networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1704.03012*, 2017.

[28] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[29] A. Yakovlev and C. Greene, "Use articulation bodies to easily prototype industrial designs with realistic motion and behavior." https://blogs.unity3d.com/2020/05/20/use-articulation-bodies-to-easily-prototype-industrial-designs-with-realistic-motion-and-behavior/, 2020. Accessed 2020-4-15.

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[31] P. Varin, L. Grossman, and S. Kuindersma, "A comparison of action spaces for learning manipulation tasks," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6015–6021, 2019.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 32, pp. 8024–8035, 2019.

[33] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[34] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3." https://github.com/DLR-RM/stable-baselines3, 2019.

[35] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." https://github.com/openai/baselines, 2017.

[36] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[37] J. Achiam, "Spinning Up in Deep Reinforcement Learning." https://spinningup.openai.com/en/latest/, 2018.

[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[39] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning (ICML)*, pp. 1889–1897, PMLR, 2015.

[40] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[41] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning (ICML)*, pp. 1928–1937, PMLR, 2016.

[42] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning (ICML)*, pp. 387–395, PMLR, 2014.

[43] S. Levine, "Cs-285: Deep reinforcement learning." http://rail.eecs.berkeley.edu/deeprlcourse-fa19/, 2019.

[44] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning (ICML)*, pp. 1587–1596, PMLR, 2018.

[45] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[46] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

[47] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 27, 2014.

[48] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *International Conference on Representation Learning (ICLR)*, 2016.

[49] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Overcoming exploration in reinforcement learning with demonstrations,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6292–6299, IEEE, 2018.

[50] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” in *International Conference on Machine Learning (ICML)*, pp. 2778–2787, PMLR, 2017.

[51] Y. Liu, P. Ramachandran, Q. Liu, and J. Peng, “Stein variational policy gradient,” in *33rd Conference on Uncertainty in Artificial Intelligence*, UAI, 2017.

[52] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller, “Data-efficient deep reinforcement learning for dexterous manipulation,” *arXiv preprint arXiv:1704.03073*, 2017.

Appendix A

Algorithm Pseudo Code

Algorithm 4 PPO

```
 1: Input: combined policy and value function parameters $\theta$, rollout buffer $\mathcal{D}$, environment E
 2: repeat
 3:     Reset the rollout buffer $\mathcal{D}$
 4:     while $|\mathcal{D}| <$ rollout_length do
 5:         Select action $a \sim \pi_\theta(\cdot|s)$
 6:         Execute $a$ in the environment
 7:         Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
 8:         Store $(s, a, r, s', d)$ in the rollout buffer $\mathcal{D}$
 9:         If $s'$ is terminal, reset environment state.
10:     end while
11:     $\theta_{old} \leftarrow \theta$
12:     Compute rollout advantage estimates $\hat{A}(s_i, a_i)$ using GAE
13:     Compute rollout return estimates $\hat{R}(s_i, a_i) = \hat{A}(s_i, a_i) + V_\theta(s_i)$
14:     Normalize the advantages
15:     for epochs iterations do
16:         for batch $B$ in SHUFFLE($\{\mathcal{D}, \hat{A}, \hat{R}\}$) do
17:             Compute the actor Loss
                $$L_t^{CLIP}(\theta, \theta_{old}, B) = \frac{1}{|B|} \sum_{(s_i,a_i)\in B} \min\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)}\hat{A}(s_i, a_i), \text{clip}\left(\frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{old}}(a_i|s_i)}, 1-\epsilon, 1+\epsilon\right)\hat{A}(s_i, a_i)\right)$$
18:             Compute the value loss
                $$L^{VF}(\theta, B) = \frac{1}{|B|} \sum_{(s_i,a_i)\in B} (\hat{R}(s_i, a_i) - V_\theta(s_i))^2$$
19:             Compute the policy entropy $S(\pi_\theta)$
20:             Compute the combined loss value
                $$L(\theta, B) = L_t^{CLIP}(\theta, \theta_{old}, B) - c_1 L^{VF}(\theta, B) + c_2 S(\pi_\theta)$$
21:             Update $\theta$ with one step of gradient ascent using
                $$\text{clip}(\nabla_\theta L(\theta, B))$$
22:         end for
23:     end for
24: until finished
```

Algorithm 5 (Twin-Delayed) DDPG [37]

```
1: Input: policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$, environment E
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: repeat
4:     Select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}(0, \sigma_{action})$
5:     Execute $a$ in the environment
6:     Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:     If $s'$ is terminal, reset environment state.
9:     if it's time to update then
10:        for $j$ in range(however many updates) do
11:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:            Compute target actions                                  ▷ noise only added for TD3
                   $$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \text{Unif}[0, \sigma_{target}]$$
                   ▷ this is an implementation bug, noise should have been $\epsilon \sim \mathcal{N}(0, \sigma_{target})$
13:            Compute targets                                          ▷ min only for TD3
                   $$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$
14:            Update Q-functions by one step of gradient descent using
                   $$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$
15:            if $j \mod \texttt{policy\_delay} = 0$ then                ▷ only for TD3
16:                Update policy by one step of gradient ascent using
                       $$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$
17:                Update target networks with
                       $$\phi_{\text{targ},i} \leftarrow (1 - \tau)\phi_{\text{targ},i} + \tau\phi_i \qquad \text{for } i = 1, 2$$
                       $$\theta_{\text{targ}} \leftarrow (1 - \tau)\theta_{\text{targ}} + \tau\theta$$
18:            end if
19:        end for
20:    end if
21: until finished
```

Appendix B

Search Hyperparameters for the Experiments from Chapter 7

This appendix lists the values of the hyperparameters that were obtained with a random search. The hyperparameters that were not included in the search are not listed here to keep the tables more compact. These values can be found in table 7.1. All experiments use the Adam optimizer [18] with default hyperparamaters as provided in Pytorch, except for the learning rate.

As already mentioned in chapter 7, for some experiments the range for Alice's entropy loss coefficient was set by accident to $[0 - 0.005]$ instead of $[0 - 0.001]$. An asterisk is put next to the entropy loss coefficients if this was the case.

Dense rewards

Bob	
Hyperparameter	**Value**
actor learning rate	0.000870
noise std deviation	0.380
batch size	128
train frequency	55/episode
hidden layer size	512
max episode duration	200

Table B.1 Hyperparameter values obtained from random search for random goal selection with dense rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.000735	actor learning rate	0.000897
		noise std deviation	0.389
		batch size	128
		train frequency	91/episode
		hidden layer size	512
max episode duration	100	max episode duration	200

Table B.2 Hyperparameter values obtained from random search for ASP with dense rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.000284	actor learning rate	0.000509
		noise std deviation	0.380
		batch size	128
		train frequency	60/episode
		hidden layer size	512
max episode duration	150	max episode duration	150

Table B.3 Hyperparameter values obtained from random search for ASP+BC with dense rewards. All other hyperparameters are listed in table 7.1.

Sparse rewards

Bob	
Hyperparameter	**Value**
actor learning rate	0.000740
noise std deviation	0.280
batch size	32
train frequency	76/episode
hidden layer size	512
max episode duration	250

Table B.4 Hyperparameter values obtained from random search for random goal selection with dense rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.000827	actor learning rate	0.000667
		noise std deviation	0.350
		batch size	64
		train frequency	75/episode
		hidden layer size	128
max episode duration	150	max episode duration	150

Table B.5 Hyperparameter values obtained from random search for ASP with sparse rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.00358*	actor learning rate	0.000745
		noise std deviation	0.380
		batch size	64
		train frequency	95/episode
		hidden layer size	128
max episode duration	100	max episode duration	200

Table B.6 Hyperparameter values obtained from random search for ASP+BC with sparse rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.00108*	actor learning rate	0.000922
		noise std deviation	0.385
		batch size	128
		train frequency	35/episode
		hidden layer size	256
max episode duration	150	max episode duration	150

Table B.7 Hyperparameter values obtained from random search for ASP + dynamic windowing with sparse rewards. All other hyperparameters are listed in table 7.1.

Alice		**Bob**	
Hyperparameter	**Value**	**Hyperparameter**	**Value**
entropy loss coefficient	0.00387*	actor learning rate	0.000883
		noise std deviation	0.397
		batch size	64
		train frequency	80/episode
		hidden layer size	256
max episode duration	150	max episode duration	150

Table B.8 Hyperparameter values obtained from random search for ASP + dynamic windowing + random goals every 4th episode with sparse rewards. All other hyperparameters are listed in table 7.1.