DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics Cognition, Intelligence

Demonstration Utilization for Long Horizon Tasks in Sparse-Reward Deep Reinforcement Learning

Mark Meinczinger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics Cognition, Intelligence

Demonstration Utilization for Long Horizon Tasks in Sparse-Reward Deep Reinforcement Learning

Verwendung von Demonstrationen in bestärkendem Lernen mit seltener Belohnung für Aufgaben mit langem Horizont

Author:	Mark Meinczinger
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Dr. rer. nat. Zhenshan Bing
Submission Date:	30.01.2023

I confirm that this master's thesis in robotics cognition, intelligence is my own work and I have documented all sources and material used.

Munich, 30.01.2023 Mark Meinczinger

Acknowledgments

I would like to express my great appreciation to my supervisor Dr. rer. nat. Zhenshan Bing, for his feedback and guidance in this project. I wish to offer my special thanks to Peter Kocsis for the discussions and for his valuable questions throughout my work.

Abstract

Using sparse reward is often preferred in reinforcement learning, thanks to avoiding the tedious work of reward engineering, while guaranteeing a well-shaped reward, by not ruling out the optimal policy for any task. On the other hand, sparse reward makes the training challenging and quite often impossible. The utilization of expert demonstrations tackles this problem by introducing an additional source of information in the form of recorded example solutions.

In this thesis, we evaluate different methods for incorporating demonstrations into the training process. We carry out a detailed comparison between previously introduced models BC (Behavior Cloning), DDPGfD (Deep Deterministic Policy Gradient from Demonstration) [1] and HinDRL (Hindsight Goal Selection for Demo-Driven RL) [2]. We evaluate our models on a series of environments based on the tasks presented in the work of Davchev et al. [2], and we also implement similar further simplified grid world environments.

Furthermore, we introduce and test the novel idea of *ϵ-filtering*, which removes over-represented demonstration states. We also show that the application of *ϵ-filtering* can stabilize the training behavior of the HinDRL [2] model.

Contents

Summary of Notation

We list the notation used by us in this summary. A lower index in vectors is used for indexing a specific element, while in functions it shows the dependency on some parameters. We omit the description of parameters specific to algorithms in this summary.

General notations:

$p(z)$	probability of random variable z
$\doteq$	equality relationship that is true by definition
$x \sim p(x)$	random variable x selected from distribution $p(x)$
$\mathbb{E}[x]$	expectation of a random variable x, i.e., $\mathbb{E}[x] \doteq \sum_x p(x)x$

Markov decision process (MDP):

ρ	function describing the dynamics of the environment
s_t	environment state at time t
r	reward function
r_t	reward at time t
a_t	action taken at time t
G_t	return or discounted (depending on the value of γ) return at time t
γ	discount factor

Reinforcement learning background:

π	policy
V^{π}	value function following policy π
V^{*}	optimal value function
Q^{π}	action-value function following policy π (also referred to as Q-function)
Q^{*}	optimal action-value function
L	loss function

Reinforcement learning algorithms:

θ	parameters used for approximating the action-value function
ψ	parameters used for approximating the value function
ϕ	parameters used in parameterized policy
R	replay buffer
$\mathcal{H}$	entropy
D_{KL}	Kullback-Leibler divergence
Z^{π}	return distribution
δ	Dirac delta function

T^{π}	Bellman operator for distributions

Goal-conditioned reinforcement learning:

g	goal state
G	goal space
r_g	goal conditioned reward function
d	some specified distance measurement
Φ	projection from state (observation) space to goal space G
S	hindsight sampling strategy
$\mathcal{U}(X)$	Uniform distribution over the set X

Demo-driven reinforcement learning:

L_{BC}	behavior cloning loss
L_{AC}	actor-critic loss
ζ	rollout buffer containing the observations, achieved goal states, rewards, and actions from one rollout
D	demonstration buffer containing the observations, achieved goal states, rewards, and actions from all demonstrations

Abbreviations

MLP	Multi Layer Perceptron
MDP	Markov Decision Process
RL	Reinforcement Learning
DQN	Deep Q-Learning [3]
DPG	Deterministic Policy Gradient [4]
DDPG	Deep Deterministic Policy Gradient [5]
SAC	Soft Actor Critic [6]
TQC	Truncated Quantile Critics [7]
GA-MDP	Goal Augmented Markov Decision Process
GCRL	Goal Conditioned Reinforcement Learning
HER	Hindsight Experience Replay [8]
HinDRL	Hindsight Goal Selection for Demo-Driven RL [2]
DDPGfD	Deep Deterministic Policy Gradient from Demonstrations [1]
EDRIAD	Efficient Reinforcement learning Insertion Approach based on Demonstrations [9]

1. Introduction

Reinforcement Learning, or more specifically, Deep Reinforcement Learning, has shown significant improvement over recent years in solving real-life tasks in robotics, like opening doors (Shixiang Gu et al. [10]), controlling a robot dog (Theodorou et al. [11]) or stacking Lego blocks and placing coat hanger on a rack only using visual input (Levine et al. [12]). Reinforcement Learning also managed to learn tasks that require long-horizon planning in highly complex environments, like playing the strategy game Dota and also managing to beat the human world champions (Berner et al. [13]).

In all types of Reinforcement Learning, the algorithm requires a reward function that evaluates the performance of the agent and can be used as an optimization target in the training process. The choice of a correct reward function is crucial in order to train an optimal policy, but in case of complex tasks, the engineering of a heuristic as a reward function is usually not straightforward and time-consuming. An incorrectly shaped reward function limits the achievable behaviors and is thus prone to learn under-performing policies. One simple way to tackle the problem of engineering a correct reward function is the application of sparse reward. Sparse reward is easy to implement, and by definition is correctly shaped, but in return, it makes the training of the policy extremely difficult. There are different methods for improving the convergence of algorithms with sparse reward. A break-through for such a solution was presented by Andrychowicz et al. in 2017 [8] with the introduction of Hindsight Experience Replay (HER), which allowed the policy to learn even from failed experiments.

Further enhancement can be reached by utilizing recorded demonstrations of an expert as an external signal for guiding the exploration of the agent as presented in the work of Vecerik et al. [9]. Contrary to reward engineering, collecting demonstrations is fairly simple and cost-efficient. Moreover, demonstrations also allow the development of algorithms that completely automatize engineering work. With these algorithms given an available simulation environment, a new task can be trained by simply recording demonstrations manually, and the training itself can be done without additional interaction. This simplification of training new tasks is an additional motivation for researching and improving these algorithms.

In this thesis, we investigate and compare different methods for incorporating demonstrations in the training procedure. In our work, we implement previously introduced models BC (Behavior Cloning), DDPGfD (Deep Deterministic Policy Gradient by Matas et al. [1]) and HinDRL (Hindsight Goal Selection for Demo-Driven RL by Davchev et al. [2]). For better comparison, we omit the implementation differences used for extracting latent observation states and rely solely on engineered observation spaces. These models use a mixture of mechanisms for utilizing expert demonstrations. In our experiments, we analyze the effectiveness of these different methods. Furthermore, we propose the method of *ϵ-filtering* for removing too close demonstration states and thus making the demonstration states evenly distributed and the training more stable. In order to perform our experiments, we implement the environments presented in the work of Davchev et al. [2] and we introduce a simplified grid-world environment as well, that can be used for fast evaluation of our models.

2. Background

In this chapter, we are going to cover the necessary background for this thesis. First of all, by introducing the methodologies of reinforcement learning in general and later by showing how hindsight samples and demonstrations can be utilized in learning complex tasks.

2.1. Reinforcement Learning

Reinforcement Learning (RL) is a computational approach for creating an agent which is capable of behaving optimally or close to optimally in its environment. In this setting, the agent has to learn to act by simply using the information gathered from past interactions. In other words, there is neither supervision for the agent nor an explicit model of the environment which could be utilized for optimal decision-making. [14]. In this section, we provide a summary of reinforcement learning and also show some specific algorithms in detail. Our aim is to cover the background of the algorithms, that we use in our experiments. For a more complete understanding of the different methods within the field of RL, we suggest reading the book *Reinforcement learning: An introduction* [14]. We also rely on this work in our summary, especially in subsection 2.1.1 and subsection 2.1.2, although we changed the notations to keep it consistent with the rest of this thesis.

2.1.1. Markov Decision Process

RL is formally described as a method tackling the problem of optimal control of incompletely-known Markov Decision Processes (MDP)[14]. MDP is a way of modeling a sequence of interactions between an acting agent and its environment. The model uses three variables

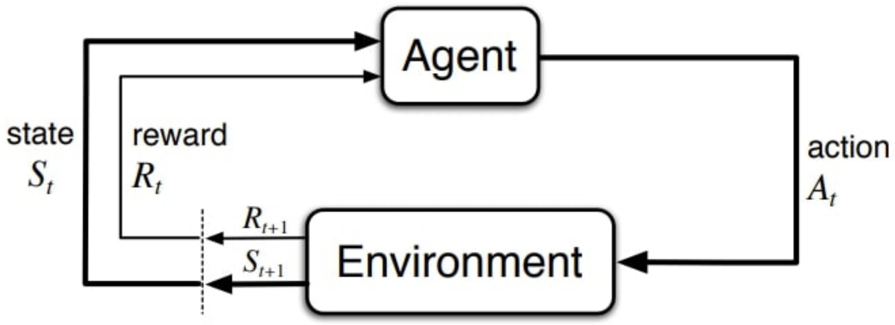


Figure 2.1.: MDP interaction model between the agent and the environment. [14]

to describe the interactions, namely the state s_t describing the environment; the action a_t

generated by the agent given the state of the environment; and the reward r_t describing the immediate "goodness" of the chosen action as shown on Figure 2.1. Using only these three variables one interaction step can be described with the tuple (s_t, a_t, r_t, s_{t+1}), and the whole sequence of interactions also referred to as episode, can be described as a collection of such tuples. In this model, we usually assume that the state of the environment after performing the action is only dependent on the current state and the chosen action. This is called the *Markov property* and can be formalized mathematically as:

$$p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}...s_0, a_0) = p(s_{t+1}|s_t, a_t) \tag{2.1}$$

In other words, the current state has to include every necessary information about the past which has an impact on the future states. Given this assumption, the dynamic of the MDP can be entirely described by the probability function $p(s_{t+1}, r_t|s_t, a_t)$. This function denoted by ρ describes the dynamics of the environment. Using only this ρ function, we are able to calculate anything of interest about the environment. In our summary, we use the following *state transition probability* function separately:

$$p(s_{t+1}|s_t, a_t) = \sum_{r_t \in \mathcal{R}} p(s_{t+1}, r_t|s_t, a_t) \tag{2.2}$$

2.1.2. Reinforcement learning background

As mentioned before RL is tackling the problem of optimal control of MDPs. As explained in subsection 2.1.1 the immediate "goodness" of an action is measured by the reward r. Our goal on the other hand is not to maximize the immediate reward of the agent but to maximize the rewards received in the long term. In general, we aim to maximize the so-called *expected return* $\mathbb{E}[G_t]$ in each action step, where the *return* G_t is simply the sum over the future rewards:

$$G_t \doteq r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.3}$$

We have to note that this formulation is only helpful if the task we want to solve is *episodic*, meaning that there is a well-defined *terminal state* for the environment, and as such G_t can be calculated as a finite sum. In *non-episodic* tasks, we have to use a modified function as a measure of optimality, namely the *discounted return*:

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.4}$$

where $0 < \gamma < 1$ is the *discount rate*. The parameter γ can be freely chosen, based on how strongly we value immediate rewards over rewards from the future. The connection between subsequent discounted returns can be simplified as follows:

$$\begin{aligned} G_t &\doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + ...) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.5}$$

Using the *discounted return*, we are able to measure how good the chosen action performed in the long term, but of course, we have to take into account that the return depends on the actions taken in the future as well. To capture this dependency, *value functions* are defined for evaluating the "goodness" of a state considering a specific way of acting, in other words following a *policy*.

The *policy* $\pi(a_t|s_t)$ can be considered as a mapping between the state s_t and chosen action a_t, or to be more precise as the probability of choosing action a_t in state s_t.

The *value function* V^π is defined as the expected return starting in state s_t and following a policy π:

$$V^\pi(s_t) \doteq \mathbb{E}\left[G_t|s_t\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t\right] \tag{2.6}$$

Similarly an *action-value function* returns the value of taking an action a_t in state s_t and following the policy π afterward:

$$Q^\pi(s_t, a_t) \doteq \mathbb{E}_\pi[G_t|s_t, a_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t, a_t\right] \tag{2.7}$$

These value functions are a core idea within RL, and the connection between the values in subsequent states can be described by the *Bellman equation*:

$$\begin{aligned} V^\pi(s_t) &\doteq \mathbb{E}_\pi[G_t|s_t] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s_t] \\ &= \sum_{a_t} \pi(a_t|s_t) \sum_{s_{t+1}, r_t} p(s_{t+1}, r_| s_t, a_t)\Big[r_t + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1}]\Big] \\ &= \sum_{a_t} \pi(a_| s_t) \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t|s_t, a_t)\Big[r_t + \gamma V^\pi(s_{t+1})\Big] \end{aligned} \tag{2.8}$$

Similarly, we can derive the *Bellman equation* for the action value function as:

$$\begin{aligned} Q^\pi(s_t, a_t) &\doteq \mathbb{E}_\pi[G_t|s_t, a_t] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s_t, a_t] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t|s_t, a_t)\Big[r_t + \gamma \mathbb{E}_{a_{t+1}\sim\pi}\big[G_{t+1}|s'_{t+1}, a_{t+1}\big]\Big] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t|s_t, a_t)\Big[r_t + \gamma \mathbb{E}_{a_{t+1}\sim\pi}[Q^\pi(s_{t+1}, a_{t+1}))]\Big] \end{aligned} \tag{2.9}$$

The goal of RL is to find the optimal policy π^* for the agent, where the optimal policy can be defined by its value function. A policy π is better than policy π' by definition if $V^\pi(s) > V^{\pi'}(s)$ for all $s \in \mathcal{S}$. Based on this observation the optimal policy is simply the policy that is better or equal to all other possible policies. Although in most cases more than one optimal policy exists for one environment, we can guarantee, based on the definition, that their optimal value function $V^*(s)$ is the same:

$$V^*(s) \doteq \max_{\pi} V^{\pi}(s) \tag{2.10}$$

Similarly, the optimal action-value function is also the same for all optimal policies:

$$Q^*(s,a) \doteq \max_{\pi} Q^{\pi}(s,a) \tag{2.11}$$

Following the definition of the action-value function (Equation 2.7) it also holds, that:

$$Q^*(s,a) = \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1})|s_t, a_t\right] \tag{2.12}$$

Since the optimal value function is also value function, we can derive the *Bellman optimality equation* for it using Equation 2.8, but in this specific case we are able to express it without using the policy:

$$\begin{aligned} V^*(s_t) &= \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s_t, a_t) \\ &= \max_{a} \mathbb{E}_{\pi^*}[G_t|s_t, a_t] \\ &= \max_{a} \mathbb{E}_{\pi^*}[r_{t+1} + \gamma G_{t+1}|s_t, a_t] \\ &= \max_{a} \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1})|s_t, a_t\right] \\ &= \max_{a} \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t|s_t, a_t)[r_t + \gamma V^*(s_{t+1})] \end{aligned} \tag{2.13}$$

Similarly, we can derive the *Bellman optimality equation* for the optimal action-value function Q^* as:

$$\begin{aligned} Q^*(s_t, a_t) &= \mathbb{E}\left[r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})|a_t, a_t\right] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t|s_t, a_t)[r_t + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})] \end{aligned} \tag{2.14}$$

The two versions of the *Bellman optimality equation* (Equation 2.13 and Equation 2.14) is the core idea for all value function based RL approach.

2.1.3. Q-learning (DQN)

There are several ways to approximate the optimal value function and the optimal policy from subsection 2.1.2 including Monte Carlo, Dynamic Programming, and other more complex methods like Temporal-Difference learning [14].

In the case of *Q-learning* introduced by Watkins and Dayan [15], we utilize the *Bellman equation* (Equation 2.9) for approximating the $Q(s,a)$ function:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &\doteq \mathbb{E}_{\pi}[G_t | s_t, a_t] \\ &= \mathbb{E}_{\pi}[r_{t+1} + \gamma G_{t+1} | s_t, a_t] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t | s_t, a_t) \Big[r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[G_{t+1} | s'_{t+1}, a_{t+1}] \Big] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t | s_t, a_t) \Big[r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^{\pi}(s_{t+1}, a_{t+1})] \Big] \end{aligned} \tag{2.9 revisited}$$

The problem with this formulation is that we need to calculate the inside expectation over the possible next actions a_{t+1}, which is often infeasible. This issue can be resolved by choosing a deterministic policy $\pi(s)$, and as a result, we can simplify the Bellman equation in the following form:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t | s_t, a_t) \Big[r_t + \gamma \mathbb{E}_{\pi}[Q^{\pi}(s_{t+1}, a_{t+1})] \Big] \\ &= \sum_{s_{t+1}, r_t} p(s_{t+1}, r_t | s_t, a_t) \Big[r_t + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1})) \Big] \end{aligned} \tag{2.15}$$

With this modification Q^{π} can be approximated only using the transitions in an offline manner, without an expectation of the actions produced by the policy. This means that Q^{π} can be learned *off-policy*. In the work of Watkins and Dayan [15], they propose the greedy policy $\pi(s) = \arg\max_a Q^{\pi}(s, a)$. Meaning, that we only need to approximate a parameterized $Q_{\theta}(s, a)$ function with parameters θ by minimizing the following loss with stochastic gradient descent:

$$\begin{aligned} L(\theta) &= \mathbb{E}\left[(Q_{\theta}(s_t, a_t) - y)^2 \right] \\ y &= r_t + \gamma Q_{\overline{\theta}}(s_{t+1}, \pi(s_{t+1})) \end{aligned} \tag{2.16}$$

Note that the target value y is also dependent on the parameter θ, but this is usually ignored during the gradient calculation step. This is in practice solved by keeping a separate *target network* $Q_{\overline{\theta}}(s, a)$ with parameters $\overline{\theta}$ which is only updated periodically with the actual network parameters $\overline{\theta} \leftarrow \theta$.

Although this method works for simple environments, more complicated scenarios and the use of large neural networks as Q-function approximators were avoided due to unstable learning. The scaling of this solution was solved by the use of *replay buffer* introduced by Mnih et al. (2013) [3]. In their solution, they stored transitions (s_t, a_t, r_t, s_{t+1}) experienced in the environment over many episodes into a so-called *replay memory* or *replay buffer*, denoted by R and utilized it as shown in Algorithm 1. By using the introduced replay buffer R, they managed to reach better sample efficiency while at the same time, they ensured a more stable training process. First by breaking the strong correlation between samples in the minibatch, and second by averaging the behavior distribution over many previous states, and as such reducing the oscillation and divergence of the parameters θ. [3]

Algorithm 1 Deep Q-learning with Experience Replay [3]

```
Initialize replay memory $R$
Initialize action-value function $Q_\theta$ with random weights $\theta$
for episode $= 1, M$ do
    Initialize sequence $S_0$
    for t=1, T do
        With probability $\epsilon$ select a random action $a_t \in \mathcal{A}(s_t)$
        otherwise select $a_t = \max_a Q_\theta(s_t, a)$
        Execute action $a_t$ in the environment
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample minibatch of transitions from $R$
        Set $y_i = \begin{cases} r_i & \text{,for terminal state } s_{i+1}. \\ r_i + \gamma \max_a Q_\theta(s_{i+1}, a) & \text{, otherwise.} \end{cases}$
        Perform a gradient descent step on $L(\theta) = (Q_\theta(s_i, a_i) - y_i)^2$
        according to Equation 2.16
    end for
end for
```

2.1.4. Deep Deterministic Policy Gradient (DDPG)

The DQN algorithm (Algorithm 1) has one flow, namely that it requires the evaluation of the max operator in Equation 2.16, which is not straightforward in the case of continuous action space. This problem can be tackled by introducing a separate parameterized policy $\pi_\phi(s)$ with parameters ϕ, also referred to as an actor; which can approximate the $\arg\max_a$ operator by learning a mapping between state s and optimal action a. This actor can substitute the greedy policy $\pi(s) = \arg\max_a Q(s,a)$ used by the DQN algorithm simply by $\pi(s) = \pi_\phi(s)$.

This so-called *actor-critic* approach was utilized in the work of Silver et al. [4] in 2014. They introduced the DPG (Deterministic Policy Gradient) algorithm, where the actor function $\pi_\phi(s)$, replacing the greedy policy, is a deterministic mapping between state s and action a. The *critic* $Q_\theta(s,a)$ can be approximated by the optimization of the loss function presented in Equation 2.16 as demonstrated before. On the other hand, we can utilize the critic to teach the actor itself as it was derived by Silver et al. [4]. The actor is updated using a gradient-based method, where the gradient of the expected return G can be calculated as follows:

$$\begin{aligned} \nabla_\phi G &\approx \mathbb{E}_{s\sim R}\Big[\nabla_\phi Q_\theta(s,a))|_{a=\pi_\phi(s)}\Big] \\ &= \mathbb{E}_s\Big[\nabla_a Q_\theta(s,a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)\Big] \end{aligned} \tag{2.17}$$

This method was developed further by Timothy P. Lillicrap et al. [5] with the introduction of DDPG (Deep Determinstic Policy Gradient), where they combined the *actor-critic* approach used in DPG with the replay buffer used in DQN for more stable training and better sample efficiency. Their method is presented in Algorithm 2.

Algorithm 2 Deep Deterministic Policy Gradient (DDPG) algorithm[5]

```
Randomly initialize critic network $Q_\theta(s,a)$ and actor $\pi_\phi(s)$ with weights $\theta$ and $\phi$
Initialize target networks $Q_{\overline{\theta}}$ and $\pi_\phi$ with: $\overline{\theta} \leftarrow \theta, \overline{\phi} \leftarrow \phi$
Initialize replay buffer $R$
**for** episode $= 1, M$ **do**
    Initialize random noise $\mathcal{N}$ for action exploration
    Receive initial observation $s_1$
    **for** t=1, T **do**
        Select action with noise: $a_t = \pi_\phi(s_t) + \mathcal{N}_t$
        Execute action $a_t$ and observe reward $r_t$ and next state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1}))$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q_{\overline{\theta}}(s_{i+1}, \pi_{\overline{\phi}}(s_{i+1}))$
        Update critic by minimizing the loss:
            $$L(\theta) = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta))^2 \qquad (2.18)$$
        Update the actor policy using the policy gradient:
            $$\nabla_\phi G \approx \frac{1}{N}\sum_i \nabla_a Q(s_i, a)|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i) \qquad (2.19)$$
        Update the target networks by *Polyak averaging*:
            $$\overline{\theta} \leftarrow \tau\theta + (1-\tau)\overline{\theta}$$
            $$\overline{\phi} \leftarrow \tau\phi + (1-\tau)\overline{\phi}$$
    **end for**
**end for**
```

The DDPG algorithm can solve tasks with continuous action spaces like the well-known Half-Cheetah task in the Mujoco [16] environment. A comprehensive comparison of different RL methods was carried out by Duan et al. [17], and their finding shows that although in simple environments DDPG can outperform other solutions both in reached score and speed of convergence, it still struggles with more complex, high dimensional tasks. The DDPG algorithm proved to be difficult to train in practice due to its brittleness and sensitivity on the hyperparameters [6].

2.1.5. Soft Actor-Critic (SAC)

In order to resolve the instability of DDPG another version of the *actor-critic* approach was introduced by Haarnoja et al. [6], where the actor behaves in a non-deterministic way. In their work, they chose the *maximum entropy* [18] as an objective function to maximize for the policy, instead of the standard expected return.

$$J_\pi = \sum_{t=0}^{T} \mathbb{E}\left[r_t + \alpha \mathcal{H}(\pi(a_t|s_t))\right] \tag{2.20}$$

The *maximum entropy* as defined in Equation 2.20 favors stochastic policies with higher entropy. The stochasticity of our target policy can be controlled with the temperature parameter α. By using this objective function, we can motivate the policy to remain stochastic and not converge to one close to optimal action, if more action can result in a similar expected return.

With this non-deterministic actor, the simplification used by DQN in Equation 2.15 does not hold anymore, so the Q-function has to be calculated by considering the effect of the policy:

$$Q^\pi(s_t, a_t) \doteq r_t + \gamma \mathbb{E}_{s_{t+1}}[V^\pi(s_{t+1})] \tag{2.21}$$

where

$$V^\pi(s_t) = \mathbb{E}_{a_t}[Q^\pi(s_t, a_t) - log\pi(a_t|s_t)] \tag{2.22}$$

is the soft value function, where the maximum entropy objective is already included by the term $-log\pi(a_t|s_t)$, considering a temperature value $\alpha = 1$. Note that contrary to the previous sections the action-value function can not be calculated without the expectation over the actions produced by the non-deterministic actor. By using the *Bellman equation* Equation 2.21 as an iterative update rule we can approximate the action-value function for any policy π as it is proved in the work of Haarnoja et al. [6]. This is the *policy evaluation* step of the SAC algorithm.

After the policy evaluation step, we update the policy based on the calculated Q-function. Since in practice usually tractable policies are favored, the policy is restricted to be from a parameterized family of distribution, usually Gaussian distribution. The goal of our policy is to get a higher expected reward measured by the action-value function. This means that we want to update our policy so that it will get closer to our target distribution $\hat{\pi}$:

$$\hat{\pi}(a_t|s_t) = \frac{exp(Q^{\pi}(s_t, a_t))}{N^{\pi}(s_t)} \tag{2.23}$$

This target distribution represents the actions with higher $Q(s, a)$ values with higher "density". Note that N^{π} is only a normalization constant, which is only necessary to get a proper target probability distribution for our policy; it does not affect the gradient calculation.

Considering the restriction on the policy as a parametric distribution, it is updated in the direction of the target distribution $\hat{\pi}$ using the Kullback-Leibler divergence:

$$\pi_{new} = \arg\min_{\pi' \in \Pi} D_{KL}\left(\pi'(a_t|s_t) \middle|\middle| \frac{exp(Q^{\pi}(s_t, a_t))}{N^{\pi}(s_t)}\right) \tag{2.24}$$

This update is referred to as the *policy improvement* step.

In the proposed method of Haarnoja et al. [6], the value function $V_{\psi}(s_t)$, the action-value function $Q_{\theta}(s_t, a_t)$ and the policy $\pi_{\phi}(a_t|s_t)$ are all approximated separately with neural networks, parameterized with ψ, θ, ϕ respectively. The constraint of parametric family for the policy $\pi_{\phi}(a_t|s_t)$ is implemented by using a neural network, which returns the mean and covariance parameters for a Gaussian distribution.

During the training we update all neural networks according to their loss functions:

$$\begin{aligned}
L_V(\psi) &= \mathbb{E}_{s_t \sim D}\left[\frac{1}{2}(V_{\psi}(s_t) - \mathbb{E}_{a_t \sim \pi_{\phi}}\left[Q_{\theta}(s_t, a_t) - log\pi_{\phi}(a_t|s_t)\right]\right] \\
L_Q(\theta) &= \mathbb{E}_{(s_t, a_t) \sim D}\left[\frac{1}{2}\Big(Q_{\theta}(s_t, a_t) - [r_t + \gamma \mathbb{E}_{s_{t+1}}\left[V_{\overline{\psi}}(s_{t+1})\right]\Big)^2\right] \\
L_{\pi}(\phi) &= \mathbb{E}_{s_t \sim D}\left[D_{KL}\left(\pi_{\phi}(a_t|s_t) \middle|\middle| \frac{exp(Q_{\theta}(s_t, a_t))}{Z_{\theta}(s_t)}\right)\right]
\end{aligned} \tag{2.25}$$

All of the loss functions in Equation 2.25 is differentiable, and as such the optimization can be solved by stochastic gradient descent. Another trick applied by Haarnoja et al. is the application of two Q-function in order to avoid positive bias resulting in an overly optimistic value function, which was originally introduced by Hado van Hasselt in 2010 [19]. The full learning process is summarized in Algorithm 3.

Algorithm 3 Soft Actor-Critic (SAC) algorithm [6]

```
Initialize the network parameters $\psi, \overline{\psi}, \theta, \phi$
Initialize replay buffer $R$
for each iteration do
    for each environment step do
        Get action: $a_t \sim \pi_\phi(a_t|s_t)$
        Perform action: $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
        Save transitions to replay buffer: $D \leftarrow D \cup \{s_t, a_t, r_t, s_{t+1}\}$
    end for
    for each gradient step do
        Update the network parameters with stochastic gradient descent:
            $\psi \leftarrow \psi - \lambda_v \nabla_\psi L_v(\psi)$
            $\theta_i \leftarrow \theta_i - \lambda_q \nabla_{\theta_i} L_q(\theta_i)$ for $i \in \{1, 2\}$
            $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi L_\pi(\phi)$
        Update the target network for the value function with Polyak update:
            $\overline{\psi} \leftarrow \tau\psi + (1-\tau)\overline{\psi}$
    end for
end for
```

2.1.6. Truncated Quantile Critics (TQC)

We already mentioned the problem of overly optimistic value functions in subsection 2.1.5, and the way how it is usually handled by learning more than one action value function as it was proposed in the work of Hado van Hasselt [19]. In this section, we are going to show a more robust solution TQC (Truncated Quantile Critics) introduced by Kuznetsov et al. in 2020 [7].

It is self-evident how important the correct estimation of the Q-function is, especially in the case when the policy optimization process depends on calculating gradients based on the action-value function, as it is done by DDPG or SAC. The work of Kuznetsov et al. [7] relies on 3 different ideas for giving a better value function estimate; namely *distributional critic*, *ensembling* and *truncation*.

Contrary to standard RL solution where only the expectation of the return is learned as a Q-function $Q^\pi(s_t, a_t) \doteq \mathbb{E}[G_t^\pi|s_t, a_t]$, the TQC algorithm takes the whole return distribution into account as a return random variable Z^π. Following the work of Dabney et al. [20] they approximate the return distribution Z^π as a mixture of M Dirac delta functions (*atoms*) at locations $(f_{\theta_1}(s_t, a_t), ...f_{\theta_M}(s_t, a_t))$. Using this representation the actual expected return can be approximated as:

$$G_t^\pi \approx \frac{1}{M}\sum_{m=1}^{M} \delta(f_{\theta_m}(s_t, a_t)) \tag{2.26}$$

where the locations are given by a parametric model f_θ. The parameter θ can be learned as previously by applying the *Bellman equation* as an update rule. This is done by minimizing the 1-Wasserstein distance between Z_θ and the target distribution $T^\pi Z_{\overline{\theta}}$, where T^π is the bellman operator for distributions. For a detailed description of the optimization process, we suggest the work of Dabney et al. [20].

The other idea of TQC is the *ensembling* of different approximators by mixing several learned atomic representations $Z_i^\pi(s_t, a_t)$ into one mixture distribution, which in effect gives a better approximation as a single approximation.

The final idea applied in the work of Kuznetsov et al. [7] is *truncation*. In order to handle the overestimation problem, they suggest a simple truncation of the right tail in the distributional critic. In the atomic representation of the critic, this can simply be done by removing some of the topmost atoms, and as such removing the overly optimistic part of the critic function.

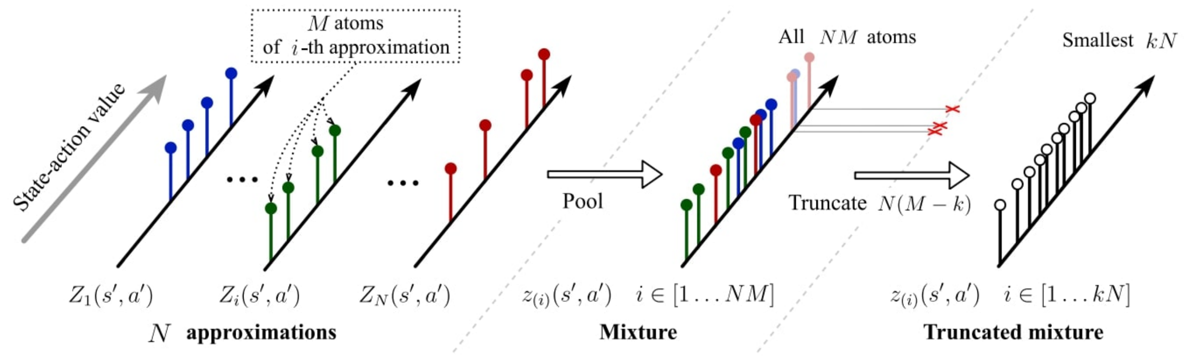


Figure 2.2.: Illustration for the calculation of the return distribution Z^π for the TQC algorithm. [7]

By utilizing these ideas and providing a better estimate of the value function, TQC was able to outperform the state-of-the-art methods on all environments from the continuous benchmark suite [17].

2.2. Goal Conditioned Reinforcement Learning

So far we did not take into consideration how the reward itself is calculated and how it influences the learned policy. In standard MDP problems (subsection 2.1.1), the reward is simply returned by the environment, and the way it is calculated is of no interest to the developer. In some specific cases on the other hand the agent has to make decisions based on an additional goal, which defines what or how the task has to be solved. This extra complexity is captured by considering a reward function $r_g(s_t, a_t, g)$ which is not only dependent on state s and action a but also on the additional goal g. As a consequence in order to choose high-rewarded action the policy $\pi(s_t, g_t) = p(a_t|s_t, g)$ itself is also has to be constrained by the defined goal. This goal-conditioned modification of the MDP is referred to as *Goal-Augmented MDP*-s (GA-MDP); and the method targeting to solve these problems is *Goal Conditioned Reinforcement Learning* (GCRL). In the GCRL setting the objective of the agent is to maximize the cumulative return:

$$G_\pi = \mathbb{E}\left[\sum_t \gamma^t r(s_t, a_t, g)\right] \tag{2.27}$$

For GA-MDP-s we have to introduce some new definitions:

Definition 1. Desired Goal: *"A desired goal is a required task to solve which can either be provided by the environments or generated intrinsically."* [21]

Definition 2. Achieved Goal: *"An achieved goal is the corresponding goal achieved by the agent at the current state."* [21]

Definition 3. Behavioral Goal: *"A behavioral goal represents the targeted task for sampling in a rollout episode."* [21]

Given these goal definitions, we can usually define an appropriate reward function in a straightforward manner, which is already able to guide the learning process of the agent for solving the task specified with the goal. Typically we can say that the problem defined by the goal and environment is solved when a predefined *criteria* is met, which can be formulated as a reward function [21]:

$$r_g(s_t, a_t, g) = \mathbb{1}(\text{the goal is reached}) \tag{2.28}$$

With this formulation, we make no assumption about the environment, and we do not constrain the possible policies in any way, so it is guaranteed that we do not rule out the optimal policy π^*. This is the so-called *sparse reward*, which is often preferred to other solutions because of the guarantee on not ruling out the optimal policy. With the usually true assumption that our criteria is met when the achieved goal is epsilon-close to the desired goal g, we can express the reward function as follows [21]:

$$r_g(s_t, a_t, g) = \mathbb{1}(||\Phi(s_{t+1}) - g|| < \epsilon) \tag{2.29}$$

where $\Phi(s)$ is the projection function extracting the achieved goal representation of state s.

Although as explained above this sparse reward is often preferred over other options, it makes the training of the policy challenging. When using a sparse reward, there is no gain for the agent from an unsuccessful episode because the reward is a constant zero for the whole sequence. The natural solution for the too sparse reward is a reshaped *dense reward*, where the reward is measured by a distance d between the achieved goal and the desired goal[21]:

$$r_g(s_t, a_t, g) = -d(\Phi(s_{t+1}), g) \tag{2.30}$$

Such dense reward signal often helps the convergence of the training process, but at the same time, it causes local optima challenges and can rule out the optimal policy by introducing additional invalid assumptions about the environment.[21]

2.2.1. Hindsight Experience Replay (HER)

As mentioned before sparse reward is often preferred over dense reward, but the training usually becomes extremely difficult because the agent is unable to learn anything useful from failed episodes. This issue was tackled by Andrychowicz et al. [8] in 2017. They proposed a method called *Hindsight Experience Replay* (HER) which is able to extract useful information even from failed rollouts and can be applied in any off-policy RL algorithm.

The usual example explaining the intuition behind their approach is an agent who plays football and has to score a goal. In standard RL with sparse reward, we would only return a non-zero reward if the agent is actually able to score. In the case of HER, we return a positive reward even if the agent is missing the gate by 1[m] on the left by considering the position "1[m] on the left" as it would have been the goal for the agent. Thus the agent can learn the dynamics of the environment and eventually will learn to score with the original goal ("the gate") as well.

In their algorithm (Algorithm 4), they collect transitions (s_t, a_t, s_{t+1}, g) in the replay buffer with the original goal g, and additionally also with an augmented goal s_t, a_t, s_{t+1}, g', where g' is a coming from an additional subset of goals G ($g' \in G$). For the augmented samples with the new goal g' the reward is calculated again in the so-called *relabeling* step. Note that although the goal is affecting the action chosen by the agent, it does not have an impact on the environment dynamics, and as such the augmented sample can be used for learning a value function without restriction. This means that we can choose a goal g' which is actually reached in the rollout based on a strategy $\mathbb{S}$, and thus we can help the training by providing a reward signal even from originally failed episodes.

Andrychowicz et al. [8] proposed 4 strategies $\mathbb{S}$ for choosing the G_t, the subset of possible goals for transition s_t:

1. **Final**: The new goal is the last state s_T reached in the rollout. ($G_t = \{s_T\}$)

2. **Future**: The new goal is a random state from the same episode as the transition, observed after the transition. ($G_t = \{s_{t+1}, s_{t+2}...s_T\}$)

3. **Episode**: The new goal is a random state from the same episode as the transition. ($G_t = \{s_0, s_1...s_T\}$)

4. **Random**: The new goal is a random state from any state encountered so far during the training. ($G_t = \{s_i \in \textit{all episode}\}$)

The algorithm introduced by Andrychowicz et al. [8] (Algorithm 4) can learn complex behaviors in practice, by utilizing sparse rewards and avoiding the local-optima issue of dense reward formulations. It is often able to outperform other methods, while at the same time no time-intensive reward engineering is required.

Algorithm 4 HER algorithm [8]

```
Given
    • an off-policy RL algorithm $\mathbb{A}$ (like DQN or SAC)
    • a strategy $\mathbb{S}$ for sampling goals for replay
    • a reward function r(s, a, g)
Initialize the neural networks for $\mathbb{A}$
Initialize replay buffer $R$
for episode $= 1, M$ do
    Sample a goal $g$ and initial state $s_0$
    for $t = 0, T - 1$ do
        Sample action $a_t$ using the behavioral policy $\pi_b$ from $\mathbb{A}$
                    $$a_t \leftarrow \pi_b(s_t, g)$$
        Execute action $a_t$, observe $s_{t+1}$
    end for
    for $t = 0, T - 1$ do
        Calculate reward for step $t$: $r_t = r(s_t, a_t, g)$
        Store transition $(s_t, a_t, r_t, s_{t+1}, g)$ in $R$
        Sample a set of additional goals $G := \mathbb{S}(\text{current episode})$
        for $g' \in G$ do
            Calculate reward for new goal $g'$ (relabeling): $r' := r(s_t, a_t, g')$
            Store transitions $s_t, a_t, r', s_{t+1}, g'$ in $R$
        end for
    end for
    for $t = 1, N$ do
        Sample a minibatch $B$ from replay buffer $R$
        Perform one-step optimization using $\mathbb{A}$ and minibatch $B$
    end for
end for
```

2.3. Demo Driven Reinforcement Learning

In the previous section (section 2.2) we covered the advantage of sparse reward and how the difficulty in training is tackled by hindsight relabeling. Although HER is able to learn complicated tasks with sparse reward settings, it still fails in complex long-horizon tasks, like the challenging towel folding task on Figure 2.3. Training a policy in such a complicated environment requires either exceptionally long training time or tedious reward engineering. One other solution is to utilize expert demonstrations, if available, as a guiding signal during the training process. In this section, we are going to introduce three approaches with the latter option.

2.3.1. Behavior Cloning

The most simple solution for incorporating information from an expert is to learn to copy its behavior precisely. This can be done by training a deterministic policy π_ϕ in a supervised manner, where the input for the policy network is state s_i and target action a_i performed by the expert in state s_i is coming from gathered demonstrations D. The policy can be optimized by minimizing the loss function:

$$L(\phi) = \mathbb{E}_{s_i,a_i \sim D}\big[|\pi_\phi(s_i) - a_i)|_2^2\big] \tag{2.31}$$

This approach is only able to perform acceptable actions in or close states present in the demonstration buffer D and so it is prone to overfitting. The other drawback of this method is that it can not outperform the expert, so it is recommended to use it only in cases when we can get enough demonstrations from a close-to-optimal expert.

2.3.2. Deep Deterministic Policy Gradient from Demonstrations (DDPGfD)

In the work of Matas et al. [1] they combined the idea of Behavior Cloning (BC) subsection 2.3.1 with the DDPG algorithm from subsection 2.1.4. In their algorithm, they initialize the replay buffer R with demonstrations D and use the actor-critic solution of DDPG [5], but they modify the loss function for the actor L_{AC} by adding an additional term with a Behavior Cloning loss L_{BC}:

$$\begin{aligned} L_{AC}(\phi) &= -\mathbb{E}_{s \sim R}\Big[Q_\theta(s,a)|_{a=\pi_{\phi(s)}}\Big] \\ L_{BC}(\phi) &= \mathbb{E}_{s_i,a_i \sim R}\big[\lambda_{BC}\delta^e|\pi_\phi(s_i) - a_i|_2^2\big] \end{aligned} \tag{2.32}$$

where λ_{BC} is the parameter for setting the importance of following the expert, and δ^e is a mask with value 1 for samples coming from demonstrations D. Their combined loss function is the following:

$$L_{actor} = L_{AC} + L_{BC} + L_{aux} \tag{2.33}$$

Where L_{aux} is an additional loss for the auxiliary outputs, which predict the key features of the environments. L_{aux} is the mean square error between the prediction and the actual value. The auxiliary loss helps the policy to learn the dynamic of the key features of the environment. This idea is necessary for their method, because they used visual input for their task. Additionally, they also use the idea of *prioratized replay* [22] and *N-step return* [23] for training the Q_θ network.

The experiments carried out by Matas et al. [5] shows, that this approach can solve the highly complex manipulation tasks presented on Figure 2.3 reliably. We also have to note that in the work of Matas et al. [5] they managed to test their agent using image observation in real-world setup.

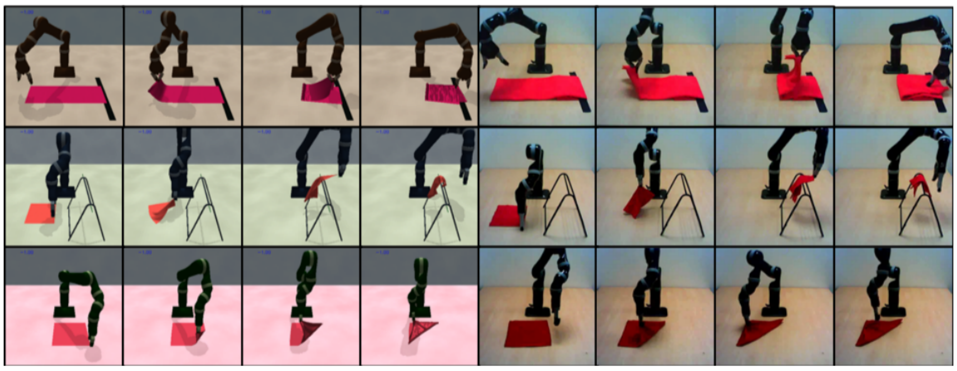


Figure 2.3.: Tasks solved by DDPGfD in the work of Matas et al: folding a large towel up to a tape (top row), hanging a small towel on a hanger (middle row) and diagonally folding a square piece of cloth (bottom row).[1]

2.3.3. HinDRL

We close our background summary with the introduction of the work of Davchev et al. [2] tackling the problem of long-horizon dexterous manipulation. We heavily rely on their work in this thesis and follow their approach with slight modifications.

Davchev et al. [2] proposed an algorithm with the main idea of hindsight relabeling constrained by gathered demonstration samples. Based on the observation that in complex environments the exploration of the whole state space is unnecessary, they suggested guiding the exploration utilizing the expert demonstrations. This is solved in the hindsight goal selection phase, where similar to HER (subsection 2.2.1) we sample new goals g' for the transitions and recalculate the reward r_t based on the distance between the new goal representation g' and the achieved goal $\Phi(s_t)$. In contrast to HER where the goal distribution $p(g')$ has non-zero probability for only the achieved goals within the same rollout ($\{g' \in \zeta : p(g') > 0\}$), Davchev et al. [2] use the achieved goals coming from the demonstrations as the goal distribution ($\{g' \in D : p(g') > 0\}$). They refer to this goal selection strategy as **Task-conditioned** hindsight sampling. This choice of goal distribution can be considered as a representation of different stages of the behavior that is required to be learned in order to solve the abstract task successfully. In the example of the cable insertion task (Figure 2.4), this means that the agent has to learn first to grab the respective cables and after that precisely bring the cable tips close to each other, orient them and finally insert them together. Failing to learn any of these sub-tasks will result in failure for the whole abstract task as well.

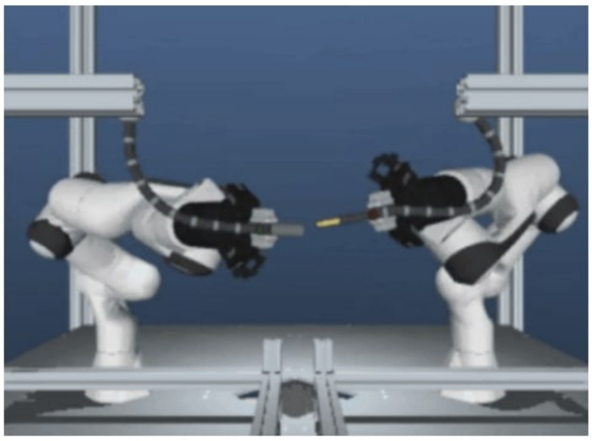


Figure 2.4.: Cable insertion task from Davchev et al. [2]

Davchev et al. [2] also experimented with different strategies for mixing rollout samples (HER samples) with the demonstration samples, and so taking advantage of learning the dynamics of the environment while at the same time rewarding the completion of stages for the actual abstract problem. They proposed the following mixed hindsight sampling strategies:

1. **Union**: Union over the rollout and demonstration data, expressed with the support: $R_G = z_g \in \zeta \cup D : p(z_g) > 0$, where ζ is containing all achieved goal from the same rollout as the transition sample.

2. **Intersection**: Intersection over the rollout and demonstration data, expressed with the support: $R_G = z_g \in \zeta \cap D : p(z_g) > 0$. Needless to say that in continuous goal spaces, this joint support would usually be empty, so instead a soft version of the intersection is applied. A sample is considered to be in both ζ and D if it is coming either from ζ or D and is ϵ-close to any sample from the other set. This results in two similar strategies depending on which set the sample is coming from.

The entire hindsight sampling pipeline is summarized on Figure 2.5.

Another trick applied by Davchev et al. [2] is the usage of BC-loss already explained in subsection 2.3.2. We also have to note that the performance of the algorithm presented by Davchev et al. [2] is highly dependent on appropriate state and goal representations, mainly because the relabeled reward is simply based on the ϵ-distance between the achieved goal and the desired goal as shown in Equation 2.29. Davchev et al. [2] experimented with both engineered and learned representations, reaching good results with both of them. In our work, we rely on engineered representations.

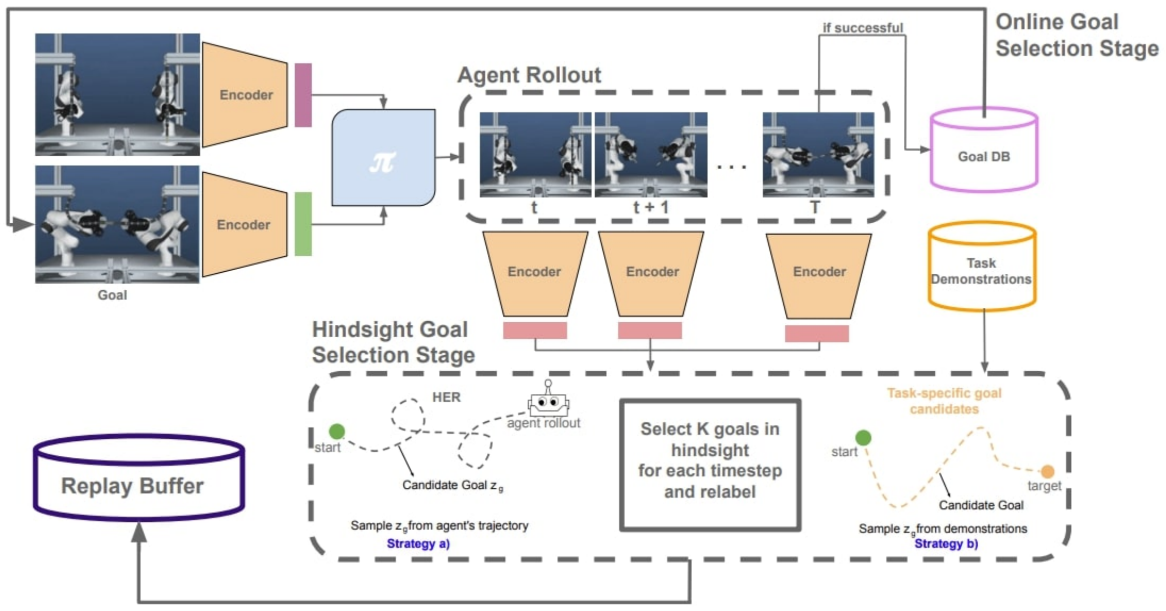

Online Goal Selection Stage
if successful
Encoder
Encoder
Goal
π
Agent Rollout
t
t + 1
T
Goal DB
Task Demonstrations
Encoder
Encoder
Encoder
Hindsight Goal Selection Stage
HER
agent rollout
start
Candidate Goal z_g
Sample z_g from agent's trajectory
Strategy a)
Select K goals in hindsight for each timestep and relabel
Task-specific goal candidates
start
target
Candidate Goal
Sample z_g from demonstrations
Strategy b)
Replay Buffer

Figure 2.5.: HinDRL hindsight sampling pipeline [2]

3. Problem statement

In our work, we compare different methods used for incorporating demonstrations in the training process. Our most complex model is based on the work of Davchev et al. [2]. We also implement the tasks presented by them for our experiments. Their approach tackles the problem of long-horizon dexterous manipulation, with their most complicated environment the *Bimanual Cable Insertion*. In this problem, two robot arm has to grab their respective cables and insert them together (Figure 2.4).

There are several reasons why this task is so complicated and why demonstration utilization is necessary to solve it. First of all, there is no straightforward way of calculating dense rewards where the local optima problem will not result in an unsuccessful policy. It can be argued that such reward calculation exist, but independently of whether this task can be solved with dense reward as well, it serves as a motivating example for looking for other solutions.

It is self-evident why a plain sparse reward method would likely never work, considering that the probability of solving this task with random actions is statistically zero, and thus there is only a constant zero reward, which can not be used for training the policy.

It can be argued that a HER-like solution could solve this task, but the usage of demonstrations is specifically efficient in this case. The efficiency of the demonstration is high first of all because the task does not change, so theoretically "overfitting" on one demonstration could solve the problem. Secondly, given the nature of the task, some subtasks are significantly easier to learn by utilizing the demonstrations. Let us consider the phase after grabbing the cables in the *Bimanual Cable Insertion* task. It is easy to see how beneficial it is to utilize the demonstrations to "overfit" the actions of the fingers, meaning that the robot should not release the cables once they are touched. This is a behavior that can easily be copied from the expert, and considering that releasing the cable even for one step would result in a failed rollout, it has a high impact on faster learning.

In our work, we also aim to solve tasks similar to the *Bimanual Cable Insertion* task, since as explained before it serves as a motivating example for sparse reward RL research. Due to the limitation in computational resources, we evaluate our models on simplified versions of this task. Given that the environments used in the work of Davchev et al. [2] are not open-source, we re-implemented them ourselves. In this chapter, we introduce the environments we created in detail. All of the environments represent similar tasks on different complexity levels starting from a grid-world representation and finally reaching the actual bimanual cable insertion task.

3.1. Grid world environments

First, we introduce a grid world environment which can be used as proof of concept for the algorithms.

3.1.1. Pick and place

Our grid world environment is a pick-and-place task, referred to as *GridPickAndPlace* in our code. This is a simple environment based on the well-known openai gym library [24].

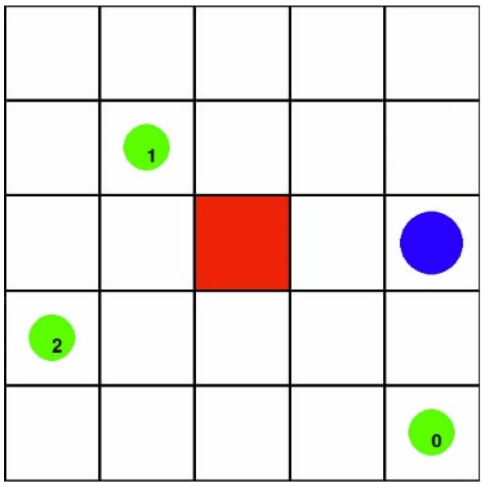


Figure 3.1.: *GridPickAndPlace* environment.

Task

The environment is demonstrated on Figure 3.1. The goal of the agent (blue dot) is to grab the objects marked with green and take them to the target position highlighted with a red rectangle. In order to do this, the agent has to move to the first object with the number 0 grab it by setting the grabbing action to 1, move it to the target rectangle while holding the grabbing action, and release the object by setting the grabbing action to 0. After that, the agent has to go for the next object and perform the same actions.

In order to keep the environment similar in nature to the original *Bimanual Cable Insertion* task, we copy some of the difficulties in the original task. One challenge in the original task is the sensitivity of keeping the cable grabbed since if the cable is released even for one step, then the whole task basically has to be started again. This difficulty is simulated by the case when the agent releases the object before reaching the target rectangle. In that case the released object will be teleported away into a random empty space.

The other difficulty is keeping the order of the subtasks. In the *Bimanual Cable Insertion* task, for example, the agent has to bring the cable heads close to each other and orient them precisely, and only after that can it start with the insertion. Skipping this subtask and going straight for the insertion is impossible and will result in a failure. Analogous to this issue the objects are marked in the order they have to be brought to the target space. If an object which is not next in the order is brought to the target it will be teleported away to a random position.

Parameters

- *size*: for value d the playing field is $d \times d$. (We use *size*=10 in our experiments.)
- *number_of_objects*: defines how many objects are in the playing field. (We use *number_of_objects*=3 in our experiments.)
- *horizon*: defines the maximum number of steps in the environment. After the allowed number of steps is exceeded, the environment will be terminated.

Observation space
Concatenation of:

- Object distance: Distance vector (2D) between the agent and all objects, stacked together into one ($2 \times$ *number_of_objects* dimensional) vector.
- Goal distance: Distance vector (2D) between the agent and the goal space.
- Transported objects: *number_of_objects* dimensional vector initialized with zeros, each index is switched to one when the respective object arrives at the goal space.
- Grabbed object: *number_of_objects* dimensional vector initialized with zeros, the value at the index of the currently grabbed object is switched to one.

Dimension: $4 \times \textit{number\_of\_objects} + 2$

Achieved goal
Equivalent to the observation space.

Action space (5D)
Concatenation of:

- Movement (4D): The index of the max value is converted to movement direction with the options "up", "down", "left", "right".
- Grabbing (1D): If the value is higher than 0.5, then the agent will grab the object which is in its current position or keep the previously grabbed object. If the value is smaller than 0.5, then the currently grabbed object is released and/or no new object will be grabbed.

Reward
The task is only considered to be solved when the last object reaches the target. Using the sparse reward framework only this step is rewarded with a reward equal to 1, all other step has a zero reward.

3.1.2. Fixed pick and place

The *GridPickAndPlace* task from subsection 3.1.1 is a simplified version of the cable insertion task with one additional difficulty, namely that the position of the objects, the agent, and the goal space are all completely random, in contrast to the cable insertion task, where the starting state is always the same. We suspect that in this setting the usage of demonstration has less impact on the training. To make the demonstration samples more informative, we created a modified version of the *GridPickAndPlace* task with the name *FixedGridPickAndPlace*.

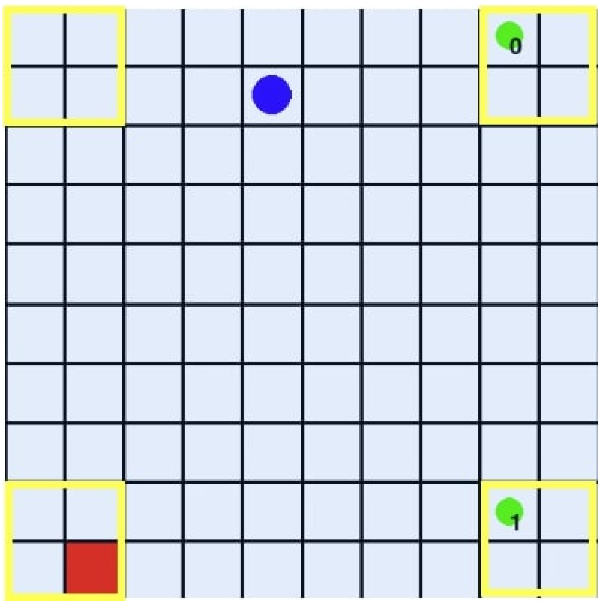
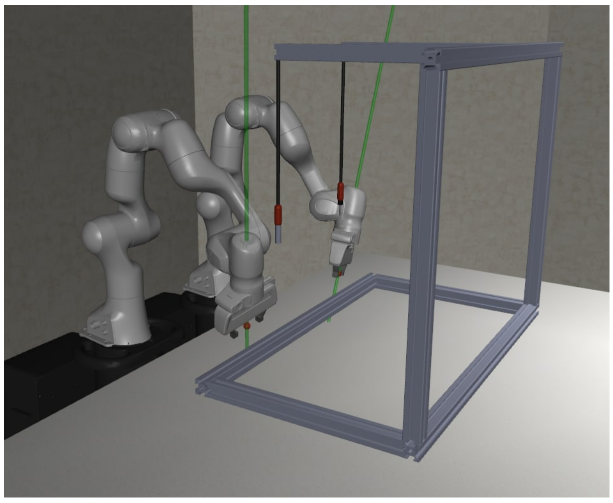


Figure 3.2.: *FixedGridPickAndPlace* environment.

The only difference we implemented into the *FixedGridPickAndPlace* environment is a constraint on the start positions for the objects, the agent, and the goal space. They are spawning into a predefined rectangle as demonstrated with the yellow mark on Figure 3.2. Additionally, if the object needs to be teleported as explained in subsection 3.1.1, then it will be teleported somewhere into its original starting rectangle. The size of the rectangle can be set with the new *random_box_size* parameter. In every other aspect, the modified environment behaves the same as the original.

3.2. Robotic environments

The next level of complexity is a continuous robotic environment. We built it based on the *Robosuite* [25] modular simulation framework and benchmark for robot learning. The framework uses the MuJoCo [16] physics engine for its simulations, and provides an OpenAI gym-like API, both of which is frequently used in RL research. The robosuite framework includes different robot arms with several options for control, different scenarios and tasks. Thanks to modularity the creation of new environments is simple. Another reason for



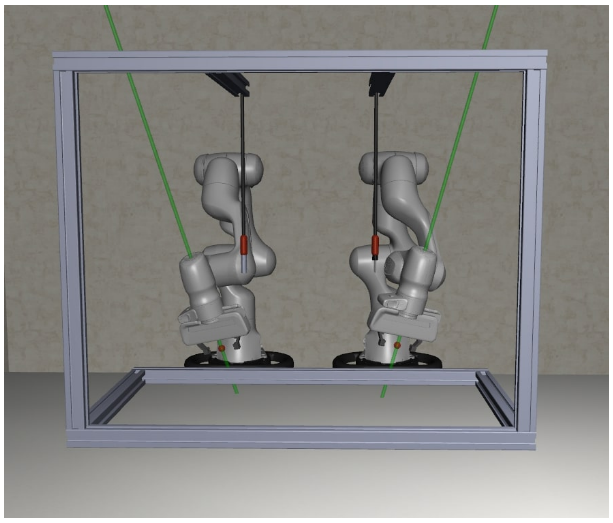


Figure 3.3.: Cable manipulation setup

choosing the robosuite framework for our work was that it already implements demonstration recording, which eased our work significantly.

Similarly to the environments presented by Davchev et al. [2], we use an *Inverse Kinematic Controller* which directly controls the relative pose for the end effectors; thus the trained policy does not have to learn the kinematics of the specified robot arm. Contrary to the environments used by Davchev et al. [2] we use a 20 Hz controller instead a 10 Hz controller, because the *Inverse Kinematic Controller* implemented in the robosuite framework behaves unstable on so low frequencies.

All of the robotic environments used in our experiments is using the same Panda robotarm illustrated on Figure 3.3. The Panda robot is a 7-DoF robot produced by Franka Emika with high positional accuracy and repeatability, which makes it suitable for a sim to real transfer in future works.

3.2.1. Parameterized Reach environment

The most simple robotic environment implemented by us is the *ParameterizedReach* environment based on the task with the same name introduced in the work of Davchev et al. [2]. In this environment, a single robot arm has to reach for different poses in a defined order as demonstrated on Figure 3.4. The task is solved when all of the poses are reached. If a pose is reached before it is the next in the required order, then it is considered to be "not reached" yet, and the robot arm has to move to it again later. After reset, the robot arm always starts from the same position, but the target poses are randomly re-sampled.

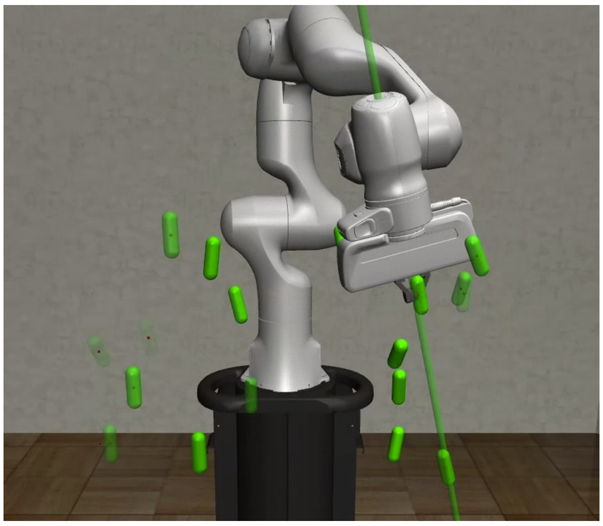


Figure 3.4.: *Parameterized Reach* environment, the target poses are marked with the green capsules. The required order of the poses is represented with the opacity of the target capsules.

Parameters

- *number_of_waypoints*: defines how many objects are in the playing field. (We use *number_of_waypoints*=2 in our experiments.)
- *horizon*: defines the maximum number of steps in the environment. After the allowed number of steps is exceeded, the environment will be terminated.

Observation space
Concatenation of:

- Global pose of the robotarm (6D): 3D for the position and 3D for the axis-angle representation of the orientation.
- Mask vector (*number_of_waypoints* $\times$ *D*): representing the already achieved goal-poses with 1 and the not reached poses with 0.

Note that the observation space does not contain any information about the waypoint poses. We only use the constrained version of this environment (subsection 3.2.2) for our experiments. For actual training in this environment, the observation space has to include information about the target poses.

Achieved goal
Equivalent to the observation space.

Action space (6D)
Concatenation of:

- Target position (x_dpos, y_dpos, z_dpos) relative to the current position of the end effector.
- Target rotation (x_rot, y_rot, z_rot) relative to the current orientation of the end effector.

The finger of the robot arm always remains closed.

Reward
The task is only considered to be solved when the last target pose is reached. The environment only returns a reward 1, when this last pose is reached, the reward is otherwise always 0.

We only use a constrained version of this task (see subsection 3.2.2) in our experiments. This environment serves only as a base for that task and future environments.

3.2.2. Fixed Parameterized Reach

Similar to the *FixedGridPickAndPlace* task we also implemented a constrained version for the *Parameterized Reach* task named *Fixed Parameterized Reach*. Based on the same difficulty assumption as for the *FixedGridPickAndPlance* task, we fix the goal poses of the *ParameterizedReach* task, so they remain the same for each initialization. This allows the models to learn the required poses solely from the demonstration. In our experiments, we only use this constrained version of the task. In every other aspect, the environment behaves the same as described in subsection 3.2.1.

3.2.3. Bring Near environment

Similar to the environment with the same name presented in the work of Davchev et al. [2], this environment is a simplification of the complete cable insertion task. The goal of the two robot arm is to grab the cables and bring the cable tips close to each other as demonstrated on Figure 3.6. In this setting, we spare the complexity of correctly orienting the cable heads and inserting them together.

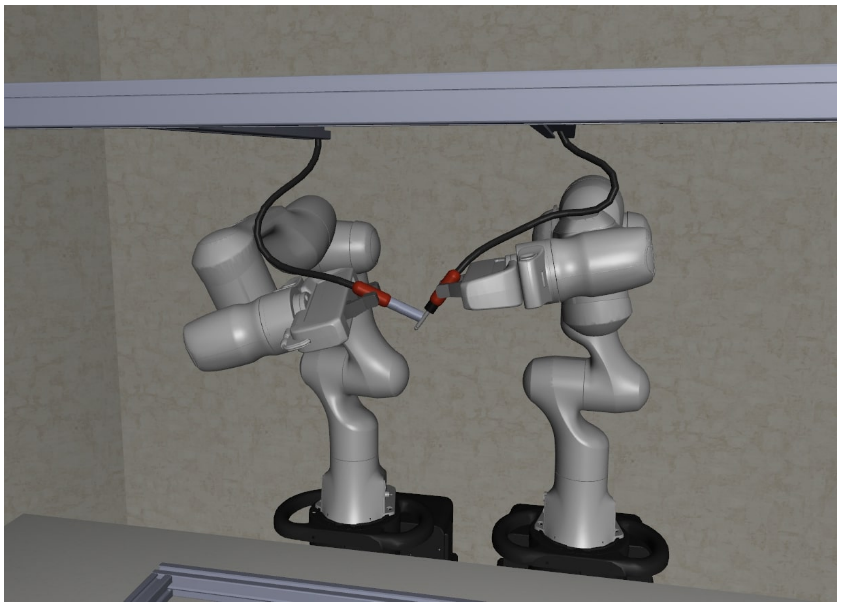


Figure 3.5.: Example solution for *BringNearEnv*

Parameters

- *horizon*: defines the maximum number of steps in the environment. After the allowed number of steps is exceeded, the environment will be terminated. In our experiments, we found that this parameter has to be chosen carefully. Too high value results in slow convergence, while a too low value results in failed training. The correct choice can depend on the algorithm used.

Observation space
Concatenation of:

- Global pose for both robotarm (12D): 3D for the position and 3D for the axis-angle representation of the orientation.
- Distance vector(3D) between the gripping site of the mother cable and its respective robot arm end-effector.
- Distance vector(3D) between the gripping site of the father cable and its respective robot arm end-effector.
- Distance vector(3D) between the cable tips.
- Grasp mother: 1 if the respective robot arm is grasping the mother cable at the gripping site, 0 otherwise.
- Grasp father: 1 if the respective robot arm is grasping the father cable at the gripping site, 0 otherwise.

- Distance (1D) between the fingers of the left robot arm.
- Distance (1D) between the fingers of the right robot arm.

Achieved goal
Equivalent to the observation space.

Action space (14D)
Concatenation of the robosuite *InverseKinematicsController* inputs for both of the robotarm. The required input is:

- Target position (x_dpos, y_dpos, z_dpos) relative to the current position of the end effector.
- Target rotation (x_rot, y_rot, z_rot) relative to the current orientation of the end effector.
- Open/close action for the fingers for both robotarms.

Reward
Given the sparse reward framework, the reward is only 1 if the cable heads are closer to each other than the defined 2[cm] threshold. If this threshold is reached, then the rollout is also terminated. In every other case, the reward is 0.

3.2.4. Cable Insertion environment

The *CableInsertionEnv* with the full bi-manual cable insertion task is the same in every aspect as the previously introduced *BringNearEnv* subsection 3.2.3, except the reward calculation.

Reward
This environment is solved when the complete cable insertion task is done. The two cables are considered to be correctly inserted if the tip of the father cable and the bottom of the mother cable is within a 3[mm] threshold and the angle between the z vectors pointing along the cable heads is smaller than 1.71° while they are pointing in opposite directions (Figure 3.6). As before, we are using sparse rewards, so the reward is only 1 when the environment is solved correctly and 0 otherwise. The environment is terminated on correct insertion. Due to limitations in the computational resources, we omit evaluation on this environment.

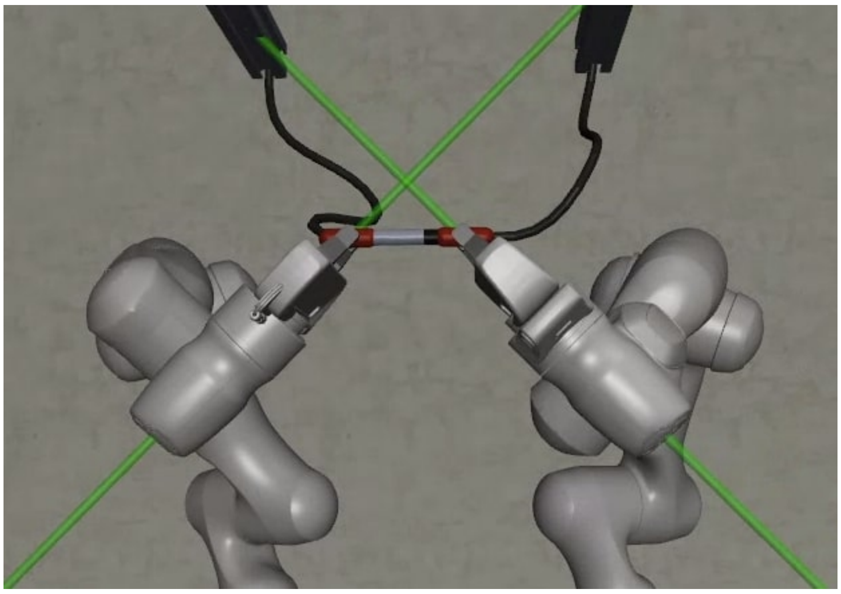


Figure 3.6.: Example solution for *CableInsertionEnv*

4. Methodology

Our goal in this thesis was to re-implement and compare different algorithms, which are tackling the problem of solving a complex sparse reward problem by utilizing demonstrations. In this chapter, we are going to explain these algorithms with some additional detail on their implementation. We also propose a novel idea called *ϵ-filtering* for boosting the performance of the HinDRL [2] model, by easing the local minima problem around the narrow passages. For keeping a clear separation between the original HinDRL model and our modified version, we will refer to it as *Smooth HinDRL*. We also differentiate the original models like HinDRL [2] and our implementation *HinDRL*, by marking all of our implemented models with an italic font.

4.1. Algorithms

We implemented 4 different algorithms:

1. Behavior Cloning (*BC*) (subsection 2.3.1)
2. Truncated Quantile Critics from Demonstrations (*TQCfD*) (based on subsection 2.3.2)
3. *HinDRL* (subsection 2.3.3)
4. *Smooth HinDRL*

In implementation algorithms 2-4. are quite similar, since *HinDRL* can be considered as a merging of the idea of *TQCfD* (based on DDPGfD [1]) and *HER* [8], with additional hindsight sampling strategies. Given this similarity, these 3 models are actually implemented as one *HinDRL* model, and we can choose the model we need with appropriate parameterization. Although *TQCfD* and *Smooth HinDRL* can be considered extreme cases of *HinDRL* we are going to refer to them separately in the future too in order to differentiate between them clearly.

In this section, we are going to first show the implementational details for the *BC* model, and secondly, explain the *HinDRL* model. Following we show how with different parametrization effectively a *TQCfD* model can be created. Finally, we introduce our new model *Smooth HinDRL*.

4.1.1. BC

The behavior cloning model as explained before in subsection 2.3.1 can be trained in the standard supervised setting where the input x is the observation and the correct output $\hat{y}$ is the action taken by the expert. We train the $\pi_\phi(x)$ "policy" by collecting demonstrations as a tuple of observation and expert actions and minimizing the loss $L(\phi)$:

$$L(\phi) = \mathbb{E}_{x_i, \hat{y}_i}\left[(\hat{y}_i - \pi_\phi(x_i))^2\right] \tag{4.1}$$

The optimization is solved with stochastic gradient descent and the optimized policy π_ϕ is an MLP network.

Parameters:

- *batch_size*: size of the minibatch
- *learning_rate*: learning rate
- *net_arch*: MLP architecture defined as a list of int-s, representing the number of perceptrons in each layer

As it is well-known input scaling is crucial for training efficiently. In order to keep the comparison with the other models fair we copy the same input processing steps. This is done in practice by extracting the policy of the same TQC module on which all of the other models are based (see subsection 4.1.2).

4.1.2. HinDRL

Our main model is the *HinDRL* based on the model introduced by Davchev et al. [2]. As explained in subsection 2.3.3, this model is a combination of DDPGfD (subsection 2.3.2) and HER (subsection 2.2.1) with a novel strategy on constraining the hindsight samples with the demonstrations. Our implementation is based on the *Stable-Baselines3* [26] framework, where several RL algorithm is readily available. The framework is set up in a modular manner which eased our work significantly. In this framework, the HER model is not actually implemented as a model but as a combination of an RL algorithm and a HER replay buffer. Utilizing this modularity, we implemented our full algorithm as a combination of four module:

1. TQC: off-policy algorithm.
2. HinDRL replay buffer: modified version of HER replay buffer.
3. Goal Handler: responsible for handling *behavioral goals* during rollout, and relabeling the hindsight samples.
4. BC-loss: additional cost value for training the policy.

We chose the TQC algorithm for two reasons. First because of the better performance (see subsection 2.1.6) and secondly because of high popularity. Thanks to the frequent usage in recent years, there are optimized hyperparameters readily available for several RL environments, so we could start our hyperparameter search based on similar tasks.

HinDRL replay buffer
We implemented our HinDRL replay buffer by adding new goal sampling strategies to the existing HER replay buffer of the *Stable-Baselines3* framework. We implemented 2 additional sampling strategies:

1. **Task-conditioned**: We sample new goals from the goal states achieved in the demonstrations by the expert. This way the policy can learn to stay close to the expert in behavior. For sampling the new goals we used a uniform distribution over all achieved-goal in the demonstrations:
$$z_g \in \mathcal{U}(D) \tag{4.2}$$
2. **Rollout-conditioned**: We sample new goals from the goal states achieved in the future within the same rollout ζ. This goal sampling is equivalent the the "future" strategy introduced with HER [8]
$$z_g \in \mathcal{U}(\zeta) \tag{4.3}$$
3. **Union**: We sample new goals from the goals achieved in the demonstrations by the expert and the goals achieved in the same rollout. This way the policy can learn to understand how the dynamics of the environment work by finding the connection between current states and states achieved in the future, while at the same time we still keep the policy similar to the expert by rewarding it additionally for following the demonstration. We use uniform distribution for sampling the goal in this case as well.
$$z_g \in \mathcal{U}(D) \cup \mathcal{U}(\zeta) \tag{4.4}$$
The ratio of goals coming from the demonstration to the goals coming from the same rollout is controlled by parameter *union_sampling_ratio*.

We omit the implementation of the *Intersection* (see subsection 2.3.3) relabeling strategy, because, as stated in the work of Davchev et al. [2], it does not improve on performance and the implementation requires *offline hindsight relabeling*. Under *offline hindsight relabeling*, we mean that the hindsight relabeling of a transition sample with a new goal is only done once and we save the resampled transition with the new goal into a buffer. Because of better memory efficiency, we chose online relabeling, where we create new goals in the hindsight relabeling process every time, when we collect a new batch for the gradient step. We also experimented with the *Intersection* strategy as online-sampling, but it made the training exceptionally slow. For this reason, we leave the implementation and evaluation of this strategy for future works.

Relabeling
For the relabeling, we need to implement an additional *compute_reward* function too for calculating the new reward value with the new sampled goal g'. Similarly to Davchev et al. [2], we added this function following the already introduced sparse reward with epsilon-distance:

$$r_g(s_t, a_t, g) = \mathbb{1}(||\Phi(s_{t+1}) - g|| < \epsilon) \tag{2.29 revisited}$$

Epsilon calculation

The relabeling method described above is sensitive to the choice of an appropriate ϵ value. In the work of Davchev et al. [2] they determine ϵ by calculating the expected distance μ^ϵ and variance σ^ϵ for achieved goals m steps away from each other:

$$\begin{aligned} \mu^\epsilon &= \mathbb{E}\,[d,t]\;||z_t^d - z_{t+m}^d|| \\ \sigma^\epsilon &= \sqrt{\mathbb{E}\,[d,t]\;(||z_t^d - z_{t+m}^d|| - \mu)^2} \end{aligned} \tag{4.5}$$

where z_t^d is the achieved goal reached in timestep t in demonstration $d \in D$.
Epsilon is finally calculated as:

$$\epsilon = \mu + k\sigma \tag{4.6}$$

where k is a freely chosen parameter.

In our work, we use a modified version of this approach. We calculate a separate ϵ_i for all dimension along the achieved goal ($z_{i,t}$):

$$\begin{aligned} \mu_i^\epsilon &= \mathbb{E}\,[d,t]\;||z_{i,t}^d - z_{i,t+m}^d|| \\ \sigma_i^\epsilon &= \sqrt{\mathbb{E}\,[d,t]\;(||z_{i,t}^d - z_{i,t+m}^d|| - \mu_i)^2} \\ \epsilon_i &= \mu_i + k\sigma_i \end{aligned} \tag{4.7}$$

Finally, we only return a reward 1 if the achieved goal z and desired z^g are closer then ϵ along all dimension:

$$r(z, z^g) = \prod_i \mathbb{1}(||\Phi(z_i - z_i^g|| < \epsilon_i) \tag{4.8}$$

With this modified epsilon calculation, we can use achieved goal spaces which differ significantly in scale for all dimensions.

BC-loss

Similar to HinDRL [2], we not only guide the policy during training with hindsight samples but also by using an additional BC-loss, to reward behavior similar to the expert. In practice, we solve this by using a demonstration buffer D separate from the replay buffer R. During training, we create our batch by sampling from our replay buffer using the strategies described above, and additionally, we add transition samples from the demonstrations to the batch as well. These demo samples are utilized in two different ways.

First, they help the critic Q_θ to learn the action-value function, by providing it with non-zero rewarded samples since all last state in the demonstration always has a reward 1. This is solely done by mixing the demonstration samples with the rollout samples. We have to note that this type of value learning can be misleading because the values for the demonstration samples are approximated for a policy that with given probability strictly follows the expert demonstrations instead of the actually trained policy π_ϕ. Nonetheless, in our experiments in section 5.3, we show that this type of demonstration utilization is capable to solve complex tasks by itself.

Second, we use the demonstration samples to give an additional BC-loss for training the actor π_ϕ. Similar to DDPGfD [1] (see subsection 2.3.2) we calculate the loss for the policy as a combination of loss coming from the critic L_{AC} and the BC-loss L_{BC}:

$$\begin{aligned} L_{AC}(\phi) &= -\mathbb{E}_{s\sim R}\Big[Q_\theta(s,a)|_{a=\pi_{\phi(s)}}\Big] \\ L_{BC}(\phi) &= \mathbb{E}_{s_i,a_i\sim R}\big[\lambda_{BC}\delta^e|\pi_\phi(s_i)-a_i|_2^2\big] \end{aligned} \tag{2.32 revisited}$$

where λ_{BC} is the parameter for setting the importance of following the expert, and δ^e is a mask with value 1 for samples coming from demonstrations D.

We decrease the number of demonstration samples over time, thus enabling the policy to learn policies not constrained by the demonstration after a given time. We simply decrease the number of demonstrations N_d linearly:

$$N_d = \max(MAX\_d(1-\frac{S}{reach\_zero}),0) \tag{4.9}$$

where MAX_d is the initial maximum number of demonstrations and *reach_zero* is the timestep when the number of demonstrations will reach and remain 0. Our whole algorithm is presented in Algorithm 5.

Algorithm 5 Our complete modified HinDRL algorithm

```
Initialize demo buffer $D$
Initialize goal database $G_{db}$ with final states from the demo buffer $D$
Calculate $\epsilon$
Initialize TQC model: actor $\pi_\phi$ and critic $Q_\theta$
Initialize replay buffer $R$
for episode $= 1, M$ do
    Get initial state: $s_0 \leftarrow p(s_0)$
    $z_g \sim G_{db}$
    $\zeta \leftarrow rollout(s_0, z^g, \pi_\phi)$
    $R \leftarrow R \cup \zeta$
    for gradient steps $= 1, N$ do
        Sample transitions from replay buffer: $B_R \sim R$
        Generate (relabeled) hindsight samples: $B_{HER}$
        Sample transitions from replay buffer: $B_{demo} \sim D$
        Create batch: $B = B_R \cup B_{HER} \cup B_{demo}$
        Update step on actor $\pi_\phi$ and critic $Q_\theta$ using batch $B$
    end for
end for
```

Parameters

- *batch_size*: size of the minibatch
- *learning_rate*: learning rate

- *net_arch*: MLP architecture defined as a list of int-s, representing the number of perceptrons in each layer
- *top_quantiles_to_drop_per_net*: top quantiles to drop in the truncation process (see subsection 2.1.6) for each approximator. (Default number of approximators is 2 and the default number of quantiles for each approximator is 25. E.g.: setting this parameter to 4, will truncate 8 quantiles out of all the 50 quantiles.)
- *buffer_size*: size of the replay buffer R
- *learning_starts*: number of environment steps to perform and load into the replay buffer R before starting the optimization
- *max_demo_ratio*: MAX_d as defined in Equation 4.9
- *reach_zero*: *reach_zero* as defined in Equation 4.9
- λ_{BC}: *reach_zero* as defined in Equation 4.9
- *n_sampled_goal*: number of additional resampled goals in the hindsight relabeling process for each sample coming from the replay buffer R. (Note: the samples coming from the demonstration buffer D are not relabeled.)
- *hindrl_sampling_strategy*: Strategy used for sampling the new goals g' in the relabeling process. Possible choices: *TaskConditioned* (HinDRL), *JointUnion* (HinDRL), *JointIntersection* (HinDRL), *RolloutConditioned* (HER "future")
- *hindrl_sampling_strategy*: Strategy used for sampling the new goals g' in the relabeling process. Possible choices: *TaskConditioned* (HinDRL), *JointUnion* (HinDRL), *JointIntersection* (HinDRL), *RolloutConditioned* (HER "future")
- *m*: *m* parameter as defined in the "Epsilon calculation" part
- *k*: *k* parameter as defined in the "Epsilon calculation" part

4.1.3. TQCfD

Our *TQCfD* model is equivalent to the DDPGfD model introduced by Matas et al. [1], with the only difference that we changed the underlying RL algorithm from DDPG [5] to TQC [7]. We also omit the implementation for the auxiliary loss, because in our work we rely on engineered observation space and so the key features are always returned by the environment itself. As mentioned before by correctly setting the parameters of our *HinDRL* model we can effectively create a *TQCfD* model too. The parameter *n_sampled_goals* defines how many samples will be relabeled using the hindsight relabeling strategy for each non-relabeled sample. Usually, in the case of *HinDRL* this parameter is set between 2 and 12 depending on the complexity of the task. By setting this parameter to 0, the hindsight relabeling is turned off entirely and so we can create a simple *TQCfD* model.

4.1.4. Smooth HinDRL

The only difference between the *HinDRL* and our *Smooth HinDRL* model is the application of *ϵ-filtering* in the goal sampling phase. This way our model can be viewed as an alternative goal sampling strategy for the original HinDRL model. A local minima issue arises in the original model, which is caused by the chosen uniform distribution for the possible goal states. This problem is illustrated on Figure 4.1.

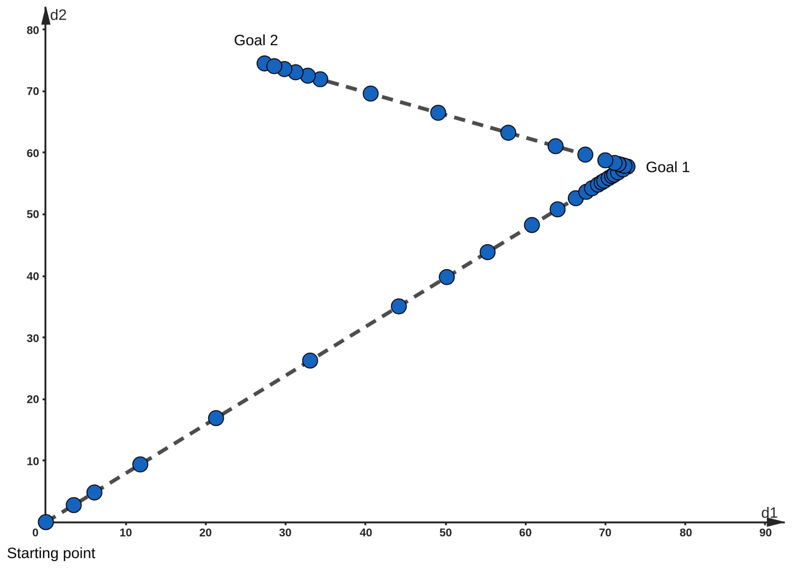


Figure 4.1.: Illustration for the local minima problem in the relabeling process. The figure above represents the demonstration for a task where two goals has to be reached in a 2D space in a specified order. The dashed line shows the trajectory of the expert, and the dots are the reached "achieved goal states" in each step. Using uniform distribution over these goal states will self-evidently result in the over-representation of areas with close states.

To tackle this issue, we either sample goal states from less "saturated" areas more often or we reduce the number of samples in the over-represented regions. We apply the latter by using an additional *ϵ-filtering* over the reached goal states in the demonstrations. We do this by removing goal samples that are too close to each other, measured by threshold ϵ (for calculation, check subsection 4.1.2). We can think about this ϵ threshold as a ball around each sample with radius ϵ. Any state inside the ball around a desired goal sample would be considered to reach the goal sample in the relabeling process. Thus all demonstration states inside a ball with ϵ-radius can be considered equivalent. We utilize this property by only keeping one from the equivalent demonstration states in the demonstration filtering step. We have to note that in our case, ϵ is not a single value, but a separate ϵ value for each dimension as explained in subsection 4.1.2. So the threshold does not represent an actual ball but a rectangular cuboid instead as shown in the Equation 4.8.

Algorithm 6 ϵ-filtering

```
Given goal database $G_{db}$
Given $\epsilon$
Initialize empty smoothed database $G_{smooth}$
for episode $E$ in $G_{db}$ do
    last_goal = None
    for $\Phi(s_t)$ in $E$ do
        if not r($\Phi(s_t)$, last_goal, $\epsilon$) (reward based on Equation 4.8) then
            $G_{smooth} \leftarrow \Phi(s_t)$
            last_goal = $\Phi(s_t)$
        end if
    end for
end for
```

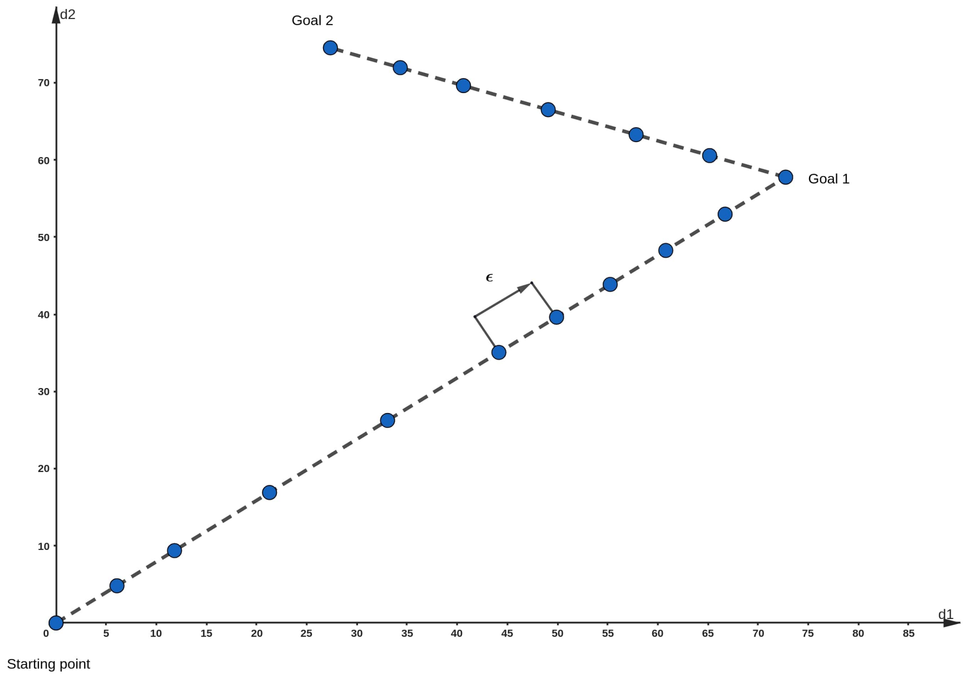


Figure 4.2.: Illustration for the *ϵ-filtered* result of the demonstration in Figure 4.1

4.2. Expert demonstrations

Expert demonstrations play a crucial role in all of the methods implemented and tested by us. Our demonstrations are simulating human-like experts, which means that they are behaving in a stochastic and non-optimal manner. The non-optimality gives room for the RL algorithm to outperform the original expert and the stochasticity results in different demonstrations, showing different ways of solving the same task.

The usage of a human-like expert is also important considering the long-term aim of the demonstration-based methods; where given a simulation environment, our goal is to train new behaviors only by providing some demonstrations created by a human without additional engineering effort. In this section, we explain and demonstrate how our experts work for each environment, and we also provide information on the implementation.

4.2.1. Grid world expert

The implemented expert for the *GridPickAndPlace* env is quite simple. Considering the two types of actions we have to realize that the task is more sensitive to the grabbing action than it is to the movement. Although when the agent does not carry any object the grabbing action does not play a role, grabbing the wrong object or releasing the good object in the wrong position increases the necessary time for solving the task significantly.

Based on this observation, our expert algorithm always uses an optimal grabbing action deterministically. This means that the grabbing action is 0 (released) as long as the agent reaches the next target object, and once the target object is reached the grabbing action is switched to 1 as long as the goal space is reached. When the goal space is reached the expert agent releases the object and continues to move in the direction of the next object.

The stochasticity and non-optimality are introduced with the *Movement* action. One wrong movement action does not have a significant impact on the whole task solution, so the expert is allowed to be more "clumsy" with this type of action. Only allowing non-optimality in the movement also simulates a human expert better. Let us consider the real-world continuous version of the *GridWorldPickAndPlace* task, a crane controlled by a human, placing boxes into a target area. The human in this case will also have a similar behavior as our non-optimal expert. The movement of the crane is going to be stochastic and non-optimal, but the grabbing and releasing can be considered to be deterministic and optimal. This type of expert also allows us to study whether our algorithms can learn some actions easier than others.

The expert movements are implemented in the following way. With a given probability defined by parameter *random_action_probability* the agent will take a completely random action every other time the agent will perform an optimal movement step. All movement is considered to be optimal if it takes the agent closer to its next goal, independently of whether the next goal is an object or the target goal space. We have to note that the optimal movement step is usually stochastic in itself, because as demonstrated on Figure 4.3, the agent can take 2 movement actions, and both of them are optimal. If more than one optimal action exists, the agent will take any of the possible actions with the same probability.

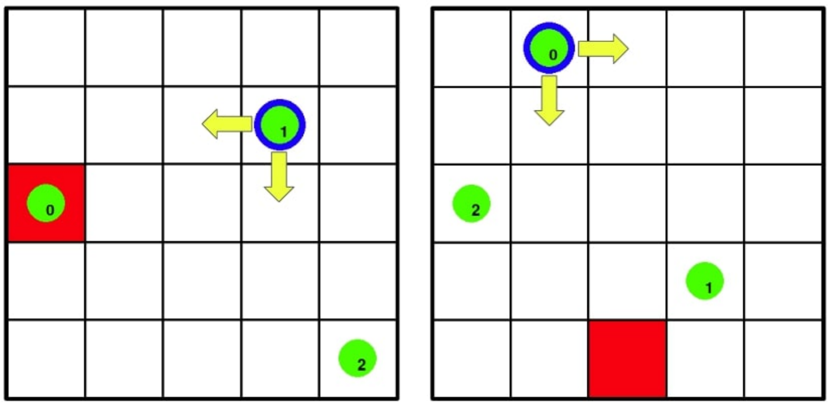


Figure 4.3.: Possible optimal movement actions for *GridWorldPickAndPlace* env.

Since the task in case of *FixedGridPickAndPlace* is practically the same as for the *GridPickAnd-Place* env, we can use the same expert to generate the demonstrations.

4.2.2. Parameterized Reach expert

The implementation of the expert for *ParameterizedReach* and *FixedParameterizedReach* is the same. The expert gets information about the pose P for the next goal (which information is not available later for the policy in the training), and simply moves the end-effector into that direction. The necessary movement and rotation values are simply defined as a difference between the current and the target pose. We introduce stochasticity into the expert demonstrations by additionally distorting the movement with a Gaussian noise e defined with the parameter *randomness_scale*:

$$\begin{aligned} e &\sim \mathcal{N}(\mu = 0, \sigma = randomness\_scale) \\ a_t &= (P_{next\_goal} - P_t) + e \end{aligned} \tag{4.10}$$

In our experiments this *randomness_scale* is set to 0.1.

4.2.3. Cable Manipulation expert

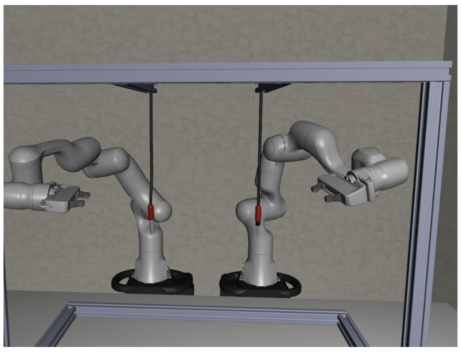
Since the robotic manipilation environments are all representing different difficulties of the full cable insertion task, we will only describe the expert algorithm for the bring near task. The only difference between the expert for this environment and the more complicated tasks is in the number of phases.

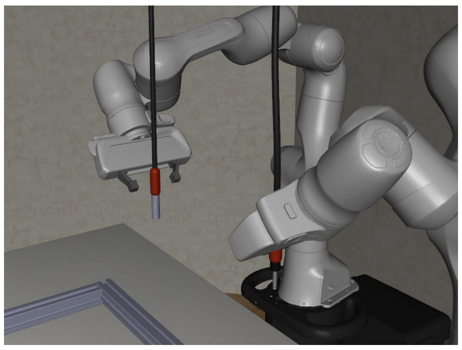
We used a similar approach for implementing the expert for the robotic environments. Based on the same idea as before, we use optimal grabbing actions while we allow more freedom for the actual movements of the robot arms. The optimal grabbing action is simply "release" until reaching the grabbing position, and "grab" afterward. Considering the movement actions, our expert has to solve the following tasks, as demonstrated on Figure 4.4:

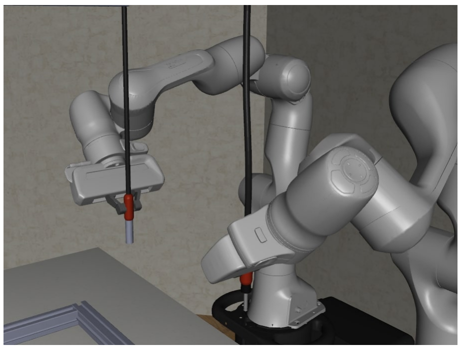
1. Get into pre-grabbing position with low precision. (Some [cm] away from the grabbable parts.)
2. Get into grabbing position with high precision.
3. Grab.
4. Wait for the other robot arm to grab the cable.
5. Bring the cable tips close to each other.

We defined the different stages with different precision, and as such, we allow more or less stochasticity in different stages. For the first stage, we sample a pose P_{target} from a relatively large box (10[cm]), in an area where it is guaranteed not to have any collision. We send the robot arm there with action $a_{movement} = P_{target} - P_{current}$. In the second stage, we send the robotarm exactly to the grabbing position. When we reached the grabbing position the robot arm grabs the cable. In the last stage, we sample a target position from a large (10[cm]) box for both cable tips. Finally, we send both robot arms there considering the distance between the cable tip and the target position. In the last step, we allow some additional freedom on the orientation of the cable tips, by selecting a random orientation within a cone around the horizontal axis.

This algorithm allows us to narrow down the acceptable state space represented by the demonstrations for specific subtasks. As a result, the policy is allowed to move more freely and discover more optimal actions in some stages, while following the demonstrations it has to learn to be precise in crucial "narrow spaces", like the grabbing.


(a) Stage 1.


(b) Stage 2.


(c) Stage 3-4.

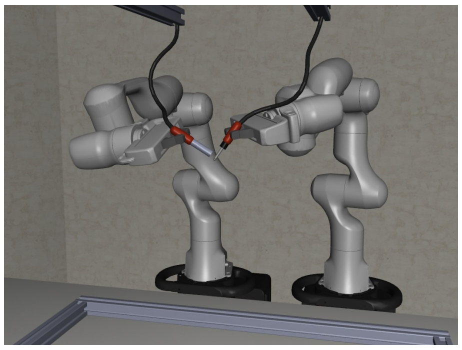

(d) Stage 5.

Figure 4.4.: Stages of the *BringNearEnv* expert

5. Experiments

In this chapter, we show our experiments and provide a detailed explanation of the results. First, we show our training results on the gridworld environments and later we present the results for the robotic environment. Due to limitations in computational resources, we only experiment with the *FixedParameterizedReach* environment from the robotic environments.

5.1. Static vs. varying task

The *GridPickAndPlace* environment provides an easy task, which enables fast evaluation and comparison of models. We mainly used this setup for proof of concept trainings, but since the training is naturally much more faster than in other environments, we also investigated some properties of our models in this setting. We have to note that in this setup, because of the grid representation, we chose the $\epsilon = 0$ value instead of approximating it as described in subsection 4.1.2. This choice also means that the ϵ-smoothing is disabled, so in this case there is no difference between the *HinDRL* and *Smoothed HinDRL* model.

As demonstrated on Table 5.1 the *HinDRL* model is clearly outperformed by the more simple *TQCfD* model. This shows that the additional hindsight samples are not helping the agent to learn a better policy.

Model comparison on *PickAndPlace* environment			
Number of demonstrations	BC success rate	TQCfD success rate	HinDRL success rate
250	75%	100%	99%
100	45%	100%	97%
50	5%	100%	4%
30	0%	4%	0%

Table 5.1.: Success rate comparison of the *BC*, *TQCfD* and *HinDRL* models on the *PickAndPlace* task. Hyperparameters for each model are available in the appendix.

We investigated the reason for this unexpected result, and it turns out that the *HinDRL* never gets a non-zero reward from the hindsight samples as demonstrated on Figure 5.1, and as a result, the only signal used for training the policy is the same BC-loss as for the *TQCfD* itself. The constant zero hindsight reward can be explained by the observation of the environment. Since the initialized position of the target objects is completely random (see subsection 3.1.1), it is self-evident that the probability of getting the same achieved goal states from different rollout is minimal, and thus reaching a resampled desired goal from a random demonstration is also unlikely. This means that the *HinDRL* model requires an environment, where the task is static and it does not change significantly for each rollout. It is worth mentioning that the task only has to be static as described by the achieved-goal states, by a correct choice of achieved-goal representation even a non-static task can be considered as static, and thus it enables the application of the *HinDRL* model.

Since, as explained above, the *PickAndPlace* task is not static, it can not be used to make a fair comparison between the models. We further evaluated our models on the constrained version of the *PickAndPlace* task, the *FixedGridPickAndPlace* task, which already can be considered static (although there is still some variability allowed in the task setup). We present our comparison on Table 5.2. As we can see, the constrained grid-world task is significantly easier to learn, even 1 demonstration can be enough for training a policy. In this simple environment the success rate of the *TQCfD* and *HinDRL* models are effectively the same, but as demonstrated on Figure 5.2 HinDRL clearly outperforms the *TQCfD* model in speed of execution.

Model comparison on *FixedPickAndPlace* environment			
Number of demonstrations	BC success rate	TQCfD s success rate	HinDRL success rate
5	17%	100%	100%
2	5%	100%	100%
1	1%	100%	100%

Table 5.2.: Success rate comparison of the *BC*, *TQCfD* and *HinDRL* models on the *FixedPickAndPlace* task. We only show the result for the converged models. Depending on the collected demonstrations and the seed 1/3 of the *TQCfD* and *HinDRL* models are unable to learn a useful policy, resulting in close to 0% success rate.

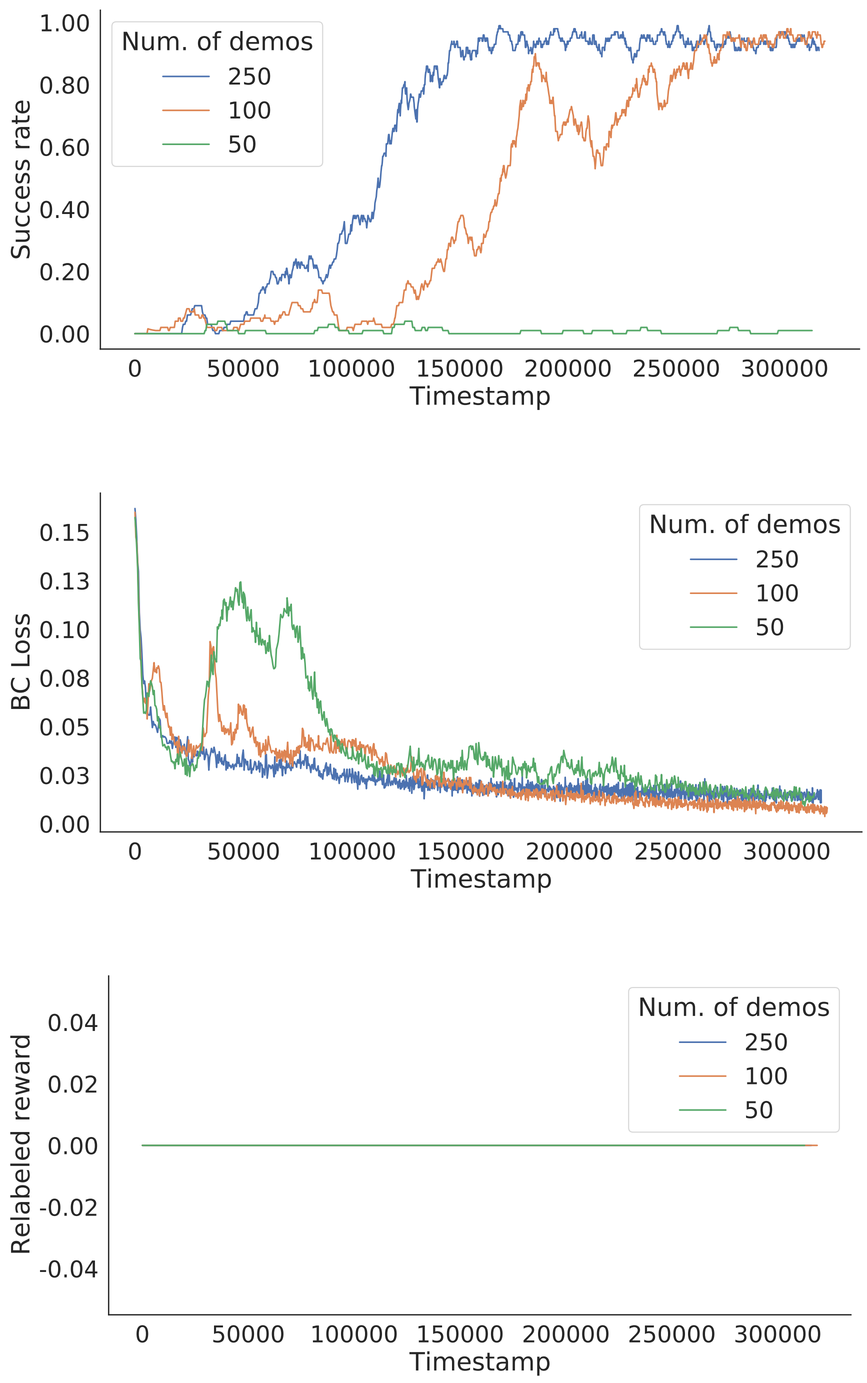
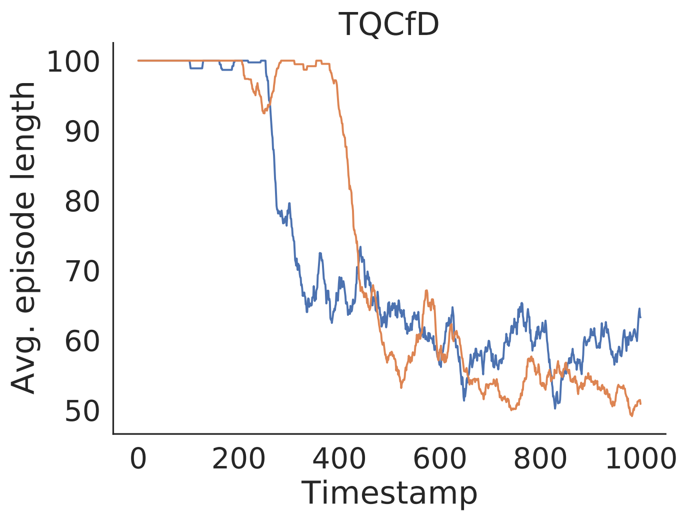
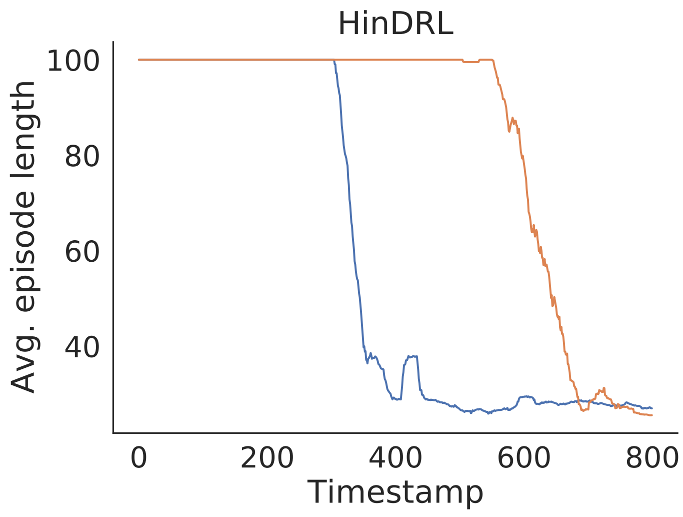


Figure 5.1.: Training results of the HinDRL model with different number of demonstrations on the *PickAndPlace* env. The first figure shows the achieved accuracy, the second the achieved BC-loss, and the last figure shows the sum of all rewards reached in the relabeling process. Note that in this environment no model is able to reach non-zero reward for the hindsight samples.



(a) Execution speed of the *TQCfD* model



(b) Execution speed of the *HinDRL* model

Figure 5.2.: Comparison of the execution speed for the HinDRL and the BC on the *Fixed-PickAndPlace* env. The different curves for each model are showing different seeds for the same training.

5.2. Continuous environment

In this section, we further investigate the effectiveness of our models in a continuous environment. This change allows us to use the non-zero ϵ value in the relabeling process, which is one of the core ideas presented by Davchev et al. for the HinDRL model. Our most simple continuous environment is the *FixedParameterizedReach*, which already satisfies the static task requirement explained in section 5.1.

Model comparison on *FixedParameterizedReach* environment			
Number of demonstrations	BC success rate	TQCfD s success rate	HinDRL success rate
100	23%	100%	100%
50	0%	100%	100%
10	0%	100%	100%
1	0%	100%	100%

Table 5.3.: Success rate comparison of the BC, TQCfD and HinDRL models on the *FixedParameterizedReach* task. The results only show the success rate for the best trained model, the distribution for the results is presented on...

Our training results summary on Table 5.3 shows that the BC model similar to the previous results is clearly outperformed by both the TQCfD and HinDRL models. As we can see both TQCfD and HinDRL is able to solve this task 100% even with 1 demonstration. Although both of these models have the same reached success rate, their performance differs both in necessary training time and the number of actually converging models.

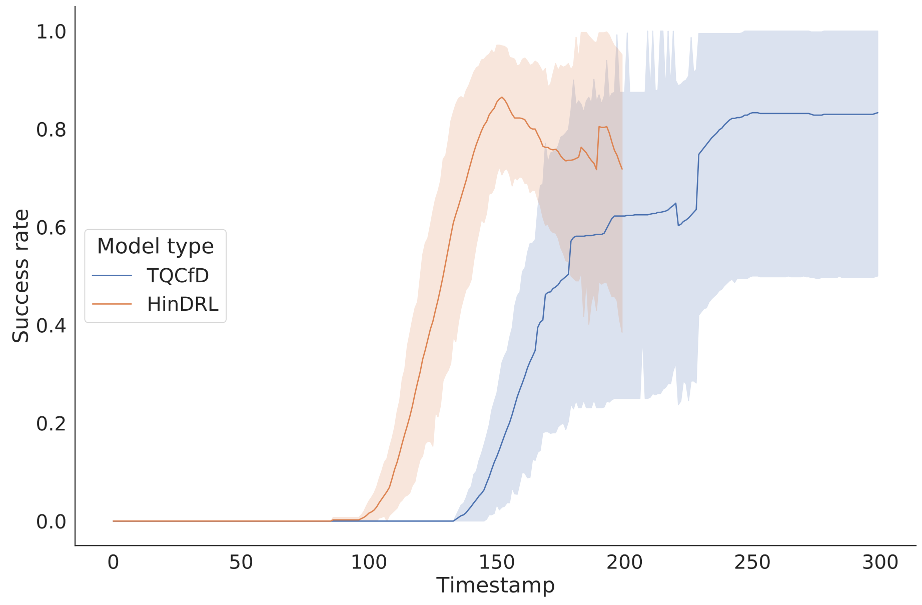
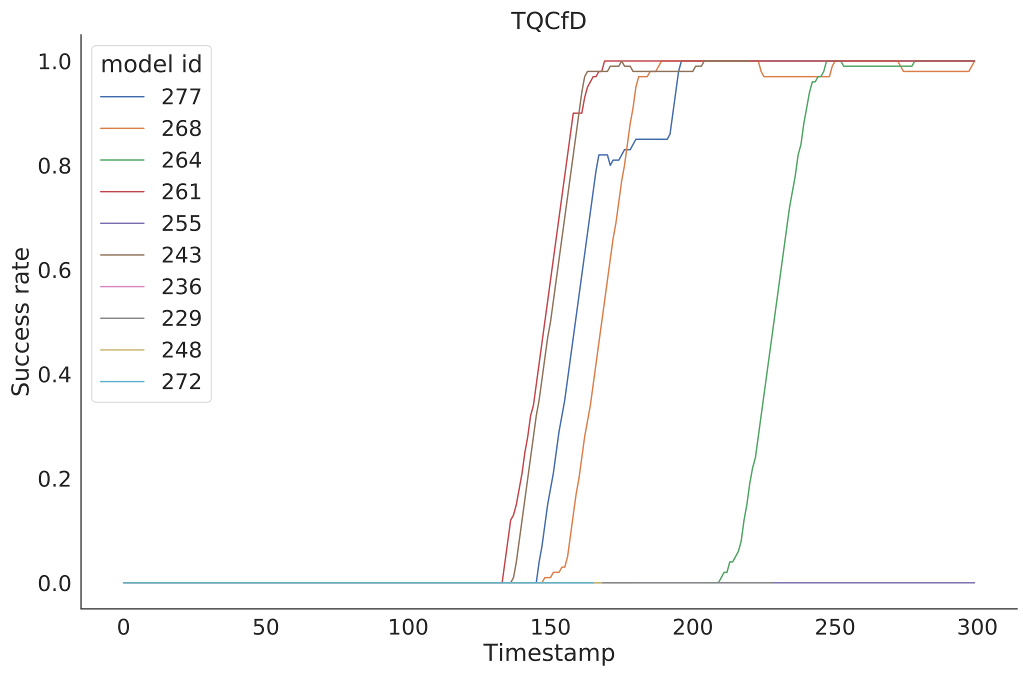


Figure 5.3.: Averaged success rates for the *TQCfD* and *HinDRL* models on the *FixedParameterizedReach* task with 1 demonstration.



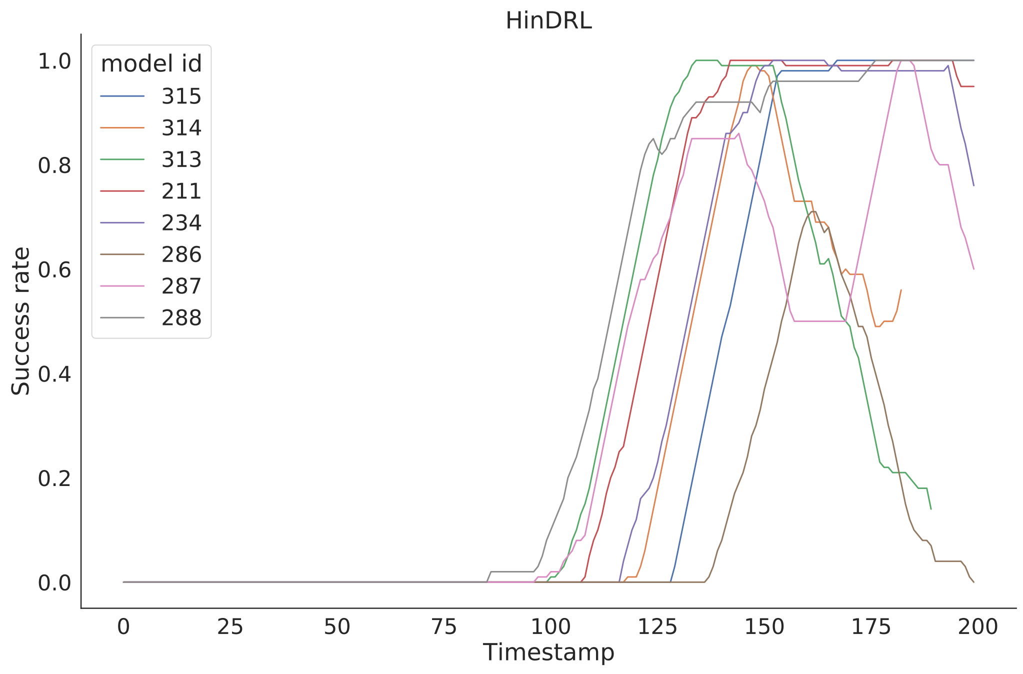


Figure 5.4.: Success rates for the *TQCfD* and *HinDRL* models showing each trained models separately on the *FixedParameterizedReach* task with 1 demonstration. All trained models within the model type had the same hyperparameters. Note that in the case of *TQCfD*, 5 out of 10 model could not solve the task at all. This is the reason for the high sigma values on Figure 5.3.

In figure Figure 5.3 we can see that the *HinDRL* model learns faster and reaches better averaged success rate in general, considering the lower sigma values around the mean curve. The actual distribution of reached success rates might be better understandable on Figure 5.4. Note that although *TQCfD* learns faster and reaches better results it suffers from an overfitting behavior, which results in a dropback in performance over time.

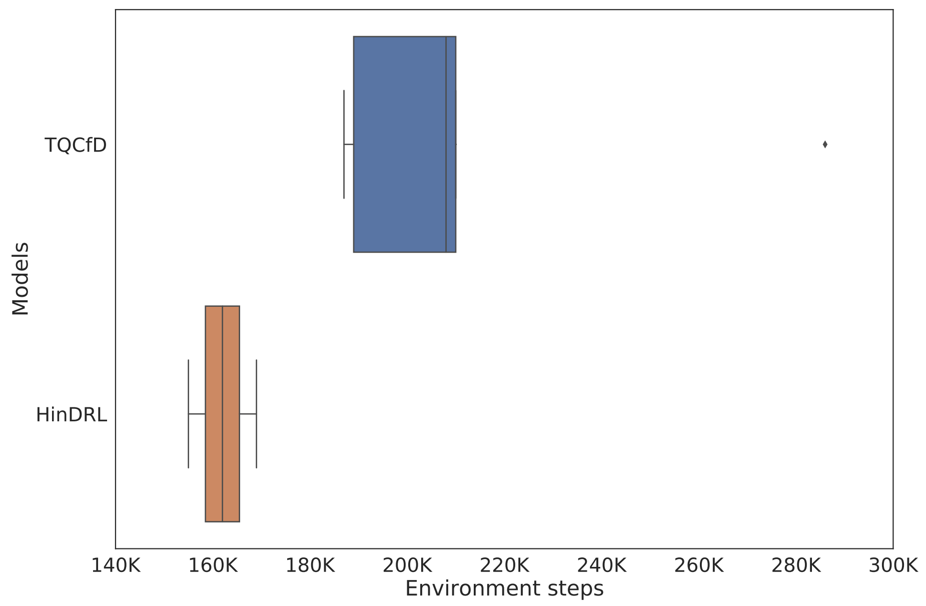


Figure 5.5.: Training time comparison for the *TQCfD* and *HinDRL* models. The figure shows the boxplots of the training times for 5 trained instances for both models. We only plot the results for the models that reached the 100% success rate.

5.2.1. Effectivity of ϵ-filtering

We evaluated the effectivity of *ϵ-filtering* for the *HinDRL* model in the *FixedParameterizedReach* environment. As presented on Figure 5.6 ϵ-smoothing does not have a significant impact on our training time and reached success rate, but it helps to stabilize the overfitting behavior of the *HinDRL* model.

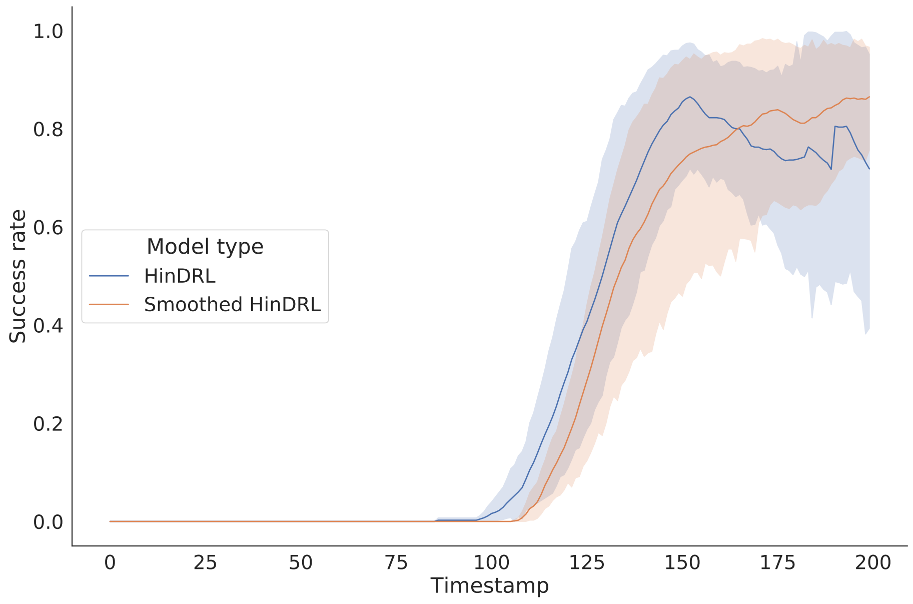


Figure 5.6.: Averaged success rates for the *HinDRL* and *Smoothed HinDRL* models on the *FixedParameterizedReach* task with 1 demonstration. Both averaged curves are generated with 8 models trained with the same hyperparameters

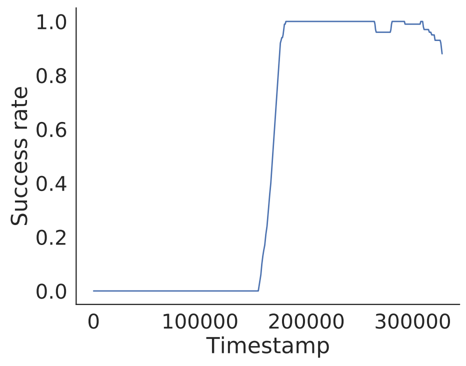
5.3. Mixing the demonstration and rollout samples

Both articles for the DDPGfD [1] and HinDRL [2] models utilize the demonstration using an additional BC-loss. Although the BC-loss guides the policy to use the actions given by the demonstration, the other training mechanisms prefer the states themselves reached in the demonstrations. Given a noisy demonstration expert these two type of training modes ("good actions vs. good states") can contradict each other. In this section, we analyze the effect of this possible contradiction.

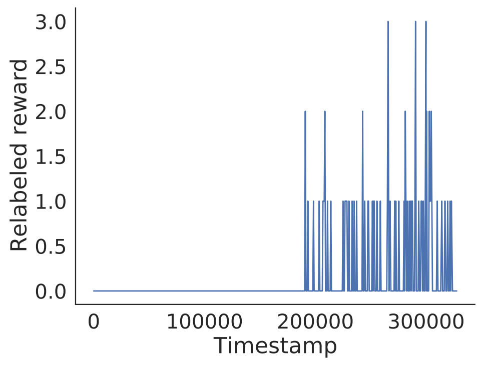
In our hyperparameter search in the continuous environment *FixedParameterizedReach* we found, that both of our models (*TQCfD* and *HinDRL*) were only able to learn the presented task with turned off BC-loss ($\lambda_{BC} = 0$). In itself, this only means that the $\lambda_{BC} = 1$ or $\lambda_{BC} = 0.1$ values used in our experiments were all too high and suppressed the other reward signals. Needless to say, we do not expect the model to converge if the BC-loss is too high, because as previously demonstrated on Table 5.3 behavior cloning is not capable of solving the *FixedParameterizedReach* environment.

We do not claim that a correct λ_{BC} parameter choice would not have a positive impact on our training, especially in more complicated environments. We only show the surprising result that both models are capable of learning with completely turned-off BC loss as well. In the case of *TQCfD* this means without a doubt that mixing demonstration samples with rollout samples is able to guide the policy on itself. We see a similar result with the *HinDRL* model since as demonstrated on Figure 5.7, the model rarely gets non-zero rewards from the hindsight samples in the starting phase of the training, and it has to rely on the same mixed demonstration sample effect as the *TQCfD*.

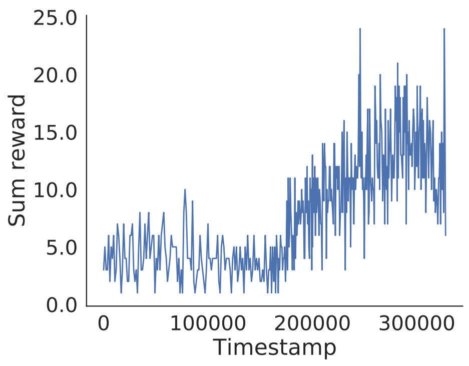
These results show that BC loss might be unnecessary in training the policy in simple environments, and mixing the demonstration samples with the rollout samples can be used on itself to solve simple tasks by utilizing demonstrations. This really simple change in the implementation allows demonstration utilization if the environment fulfills the "static task" requirement from section 5.1. A similar model was previously introduced by Liu et al. in 2022 [27]. Further application of the BC loss could still improve the performance of the policy, but requires the careful setting of the λ_{BC} parameter.



(a) Success rate during the training.



(b) Sum of the hindsight rewards during the training.



(c) Sum of all rewards during the training.

Figure 5.7.: Training details for an example *HinDRL* model with turned off BC-loss on the *FixedParameterizedReach* environment with 1 demonstrations. 2nd and 3rd figures show the rewards experienced during the training process. The *batch_size* for this model is 12000. Having the value 10 in all rewards (figure (c)) and 2 in the hindsight rewards (figure (b)) at a training step means, that out of 12000 sample 8 had a non-zero reward (successful final state) and 2 got a reward 1 in the hindsight relabeling process.

6. Future work

In our experiments, we did not evaluate our models in the more complicated environments due to too long training times. We suspect that the difference in performance would be more significant between the *Smoothed HinDRL* and *HinDRL* models. We leave this evaluation for the future.

We omit showing the comparison of the different resampling strategies presented in the HinDRL [2] paper and we only show results for the *Task-conditioned* strategy. We performed some training with the *Union* strategy too, but it was clearly outperformed by the other method as also described in the original paper. For this reason and because of time limitations we did not optimize the hyperparameters for this strategy and so we do not show our training results either. We think that the idea of the *Union* strategy, namely resampling rollout goals to learn the dynamics of the environment, while also guiding the exploration with demonstration goals, should perform better. We suspect that the reason for the bad performance is based on the online goal generation step. In our work and also in the HinDRL paper [2] the behavioral goal used as a goal during the rollout generation is always a terminal goal state of a demonstration. This hinders the model in learning the dynamics of the environment. We suspect that allowing the model to use goals in the rollout generation phase from previously achieved states would improve its performance, by enabling it to discover the environment more freely. We consider this modification a promising direction for research.

We modified both the HinDRL [2] and DDPGfD [1] in implementation, by switching the underlying RL algorithm from DDPG [5] to TQC [7]. This modification was only done because of practical reasons, since TQC was more popular in recent years and so served as a good base for our hyperparameter search. We did not expect an actual performance difference from this change, but our results show slightly better performance than reported in the original papers. Our environments slightly differ from the ones used in the HinDRL article [2] too, so no clear conclusion can be made about the effect of this change. Nonetheless, we suspect that the application of TQC has a positive impact on our results. In order to prove this suspicion further comparison is necessary between the underlying DDPG and TQC models.

The aim of the research of demonstration utilization methods is to eventually solve tasks in real-life scenarios. In this thesis, we omit the implementation of relevant feature extraction and rely on engineered observation space. In real-life tasks, this simplification does not hold, and a feature extractor plays a crucial role in the actual performance. Some works like Matas et al. [1], Vecerik et al. [9] and Davchev et al. [2] already tackled this problem. Future comparison has to be carried out also including the differences of the feature extractor to cover the whole problem of demonstration utilization.

7. Conclusion

In this thesis, we compared different methods for including gathered demonstrations in the training process to boost our models' training time and performance. We implemented 4 distinct model the standard *BC*, *TQCfD* (based on DDPGfD [1]), *HinDRL* (based on the work of Davchev et al. [2]) and *Smoothed HinDRL*, by implementing and additional *ϵ-filtering* step for the demonstrations, which removes over-represented demonstration states. We evaluated these models on the varying task of *GridPickAndPlance* and compared it to the static task of *FixedGridPickAndPlance*. We further compared our models on the simple robotic task *FixedParameterizedReach*. We also implemented more complicated environments following the same tasks as presented in the work of Davchev et al. [2].

We demonstrate the necessary requirement of tasks to remain *static*, which has to be fulfilled for demonstration utilization. We present that *Smoothed HinDRL* shows more stable training behavior, while both *HinDRL* and *Smoothed HinDRL* clearly outperforms the two more simple models.

Furthermore, we prove, that surprisingly both *TQCfD* and *HinDRL* model is capable of learning simple tasks solely relying on mixing demonstrations with the rollout samples in the early stages of training. Although *HinDRL* outperforms the more simple *TQCfD* model, this shows the opportunity, to utilize demonstrations in any actor-critic algorithm by simply mixing demonstration samples into the rollout buffer. Since this can be done with low implementational effort, it can be beneficial in many applications.

A. Appendix

In this chapter, we present the detailed list of the hyperparameters used in our experiments.

A.1. Hyperparameters

A.1.1. Gridworld environments

Hyperparameters used in the experiments in section 5.1 for the *GridPickAndPlace* and the *FixedGridPickAndPlace* tasks:

BC model hyperparameters	
Hyperparameter	Value
$batch\_size$	2048
$learning\_rate$	$1e^{-3}$
net_arch	$[32, 32]$

TQCfD model hyperparameters	
Hyperparameter	Value
$batch\_size$	4096
$learning\_rate$	$1e^{-3}$
net_arch	$[64, 64, 64, 64]$
$top\_quantiles\_to\_drop\_per\_net$	2
$buffer\_size$	$5e^{5}$
$learning\_starts$	$1e^{4}$
λ_{BC}	1
max_demo_ratio	0.1
$reach\_zero$	$5e^{5}$
$n\_sampled\_goal$	0
$hindrl\_sampling\_strategy$	*TaskConditioned* (not used)
m	0 (not used)
k	0 (not used)
$epsilon\_filtering$	*False*

HinDRL model hyperparameters	
Hyperparameter	Value
$batch\_size$	4096
$learning\_rate$	$1e^{-3}$
net_arch	$[64, 64, 64, 64]$
$top\_quantiles\_to\_drop\_per\_net$	2
$buffer\_size$	$5e^{5}$
$learning\_starts$	$1e^{4}$
λ_{BC}	1
max_demo_ratio	0.1
$reach\_zero$	$5e^{5}$
$n\_sampled\_goal$	4
$hindrl\_sampling\_strategy$	*TaskConditioned*
m	0
k	0
$epsilon\_filtering$	*False*

A.1.2. Continuous environments

Hyperparameters used in the experiments in section 5.2 for the *FixedParameterizedReach* task:

BC model hyperparameters	
Hyperparameter	Value
$batch\_size$	2048
$learning\_rate$	$1e^{-3}$
net_arch	$[32, 32]$

TQCfD model hyperparameters	
Hyperparameter	Value
batch_size	$1.2e^4$
learning_rate	$1e^{-3}$
net_arch	$[32, 32, 32, 32, 32, 32, 32, 32]$
top_quantiles_to_drop_per_net	8
buffer_size	$1e^6$
learning_starts	5000
λ_{BC}	0
max_demo_ratio	0.1
reach_zero	$1e^6$
n_sampled_goal	0
hindrl_sampling_strategy	*TaskConditioned* (not used)
m	0 (not used)
k	0 (not used)
epsilon_filtering	*False*

HinDRL model hyperparameters	
Hyperparameter	Value
batch_size	$1.2e^4$
learning_rate	$1e^{-3}$
net_arch	$[32, 32, 32, 32, 32, 32, 32, 32]$
top_quantiles_to_drop_per_net	8
buffer_size	$1e^6$
learning_starts	5000
λ_{BC}	0
max_demo_ratio	0.1
reach_zero	$1e^6$
n_sampled_goal	4
hindrl_sampling_strategy	*TaskConditioned*
m	5
k	0
epsilon_filtering	*False*

Smoothed HinDRL model hyperparameters	
Hyperparameter	Value
$batch\_size$	$1.2e^4$
$learning\_rate$	$1e^{-3}$
net_arch	$[32, 32, 32, 32, 32, 32, 32, 32]$
$top\_quantiles\_to\_drop\_per\_net$	8
$buffer\_size$	$1e^6$
$learning\_starts$	5000
λ_{BC}	0
max_demo_ratio	0.1
$reach\_zero$	$1e^6$
$n\_sampled\_goal$	4
$hindrl\_sampling\_strategy$	*TaskConditioned*
m	5
k	0
$epsilon\_filtering$	*True*

List of Figures

List of Tables

Bibliography

[1] J. Matas, S. James, and A. J. Davison. “Sim-to-Real Reinforcement Learning for Deformable Object Manipulation”. In: *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*. Vol. 87. Proceedings of Machine Learning Research. PMLR, 2018, pp. 734–743. URL: `http://proceedings.mlr.press/v87/matas18a.html`.

[2] T. Davchev, O. O. Sushkov, J. Regli, S. Schaal, Y. Aytar, M. Wulfmeier, and J. Scholz. “Wish you were here: Hindsight Goal Selection for long-horizon dexterous manipulation”. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: `https://openreview.net/forum?id=FKp8-pIRo3y`.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. Vol. 32. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 387–395. URL: `http://proceedings.mlr.press/v32/silver14.html`.

[5] 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”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2016. URL: `http://arxiv.org/abs/1509.02971`.

[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1856–1865. URL: `http://proceedings.mlr.press/v80/haarnoja18b.html`.

[7] A. Kuznetsov, P. Shvechikov, A. Grishin, and D. P. Vetrov. “Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5556–5566. URL: `http://proceedings.mlr.press/v119/kuznetsov20a.html`.

[8] M. Andrychowicz, D. Crow, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. “Hindsight Experience Replay”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett. 2017, pp. 5048–5058. URL: `https://proceedings.neurips.cc/paper/2017/hash/453fadbd8a1a3af50a9df4df899537b5-Abstract.html`.

[9] M. Vecerik, O. Sushkov, D. Barker, T. Rothorl, T. Hester, and J. Scholz. “A Practical Approach to Insertion with Variable Socket Position Using Deep Reinforcement Learning”. In: *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24, 2019*. IEEE, 2019, pp. 754–760. DOI: `10.1109/ICRA.2019.8794074`. URL: `https://doi.org/10.1109/ICRA.2019.8794074`.

[10] S. Gu, E. Holly, T. P. Lillicrap, and S. Levine. “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates”. In: *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*. IEEE, 2017, pp. 3389–3396. DOI: `10.1109/ICRA.2017.7989385`. URL: `https://doi.org/10.1109/ICRA.2017.7989385`.

[11] E. A. Theodorou, J. Buchli, and S. Schaal. “Reinforcement learning of motor skills in high dimensions: A path integral approach”. In: *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*. IEEE, 2010, pp. 2397–2403. DOI: `10.1109/ROBOT.2010.5509336`. URL: `https://doi.org/10.1109/ROBOT.2010.5509336`.

[12] S. Levine, C. Finn, T. Darrell, and P. Abbeel. “End-to-End Training of Deep Visuomotor Policies”. In: *J. Mach. Learn. Res.* 17 (2016), 39:1–39:40. URL: `http://jmlr.org/papers/v17/15-522.html`.

[13] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. “Dota 2 with Large Scale Deep Reinforcement Learning”. In: *CoRR* abs/1912.06680 (2019). arXiv: `1912.06680`. URL: `http://arxiv.org/abs/1912.06680`.

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

[15] C. J. C. H. Watkins and P. Dayan. “Technical Note Q-Learning”. In: *Mach. Learn.* 8 (1992), pp. 279–292. DOI: `10.1007/BF00992698`. URL: `https://doi.org/10.1007/BF00992698`.

[16] 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*. 2012, pp. 5026–5033. DOI: `10.1109/IROS.2012.6386109`.

[17] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by M. Balcan and K. Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 1329–1338. URL: `http://proceedings.mlr.press/v48/duan16.html`.

[18] B. D. Ziebart. "Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy". PhD thesis. Carnegie Mellon University, USA, 2010. DOI: `10.1184/r1/6720692.v1`. URL: `https://doi.org/10.1184/r1/6720692.v1`.

[19] H. van Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*. Ed. by J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta. Curran Associates, Inc., 2010, pp. 2613–2621. URL: `https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html`.

[20] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos. "Distributional Reinforcement Learning With Quantile Regression". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press, 2018, pp. 2892–2901. URL: `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17184`.

[21] M. Liu, M. Zhu, and W. Zhang. "Goal-Conditioned Reinforcement Learning: Problems and Solutions". In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Ed. by L. D. Raedt. ijcai.org, 2022, pp. 5502–5511. DOI: `10.24963/ijcai.2022/770`. URL: `https://doi.org/10.24963/ijcai.2022/770`.

[22] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. "Prioritized Experience Replay". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun. 2016. URL: `http://arxiv.org/abs/1511.05952`.

[23] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap. "Distributed Distributional Deterministic Policy Gradients". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: `https://openreview.net/forum?id=SyZipzbCb`.

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[25] Y. Zhu, J. Wong, A. Mandlekar, and R. Martin-Martin. "robosuite: A Modular Simulation Framework and Benchmark for Robot Learning". In: *CoRR* abs/2009.12293 (2020). arXiv: `2009.12293`. URL: `https://arxiv.org/abs/2009.12293`.

[26] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *J. Mach. Learn. Res.* 22 (2021), 268:1–268:8. URL: `http://jmlr.org/papers/v22/20-1364.html`.

[27] H. Liu, Z. Huang, J. Wu, and C. Lv. "Improved Deep Reinforcement Learning with Expert Demonstrations for Urban Autonomous Driving". In: *2022 IEEE Intelligent Vehicles Symposium, IV 2022, Aachen, Germany, June 4-9, 2022*. IEEE, 2022, pp. 921–928. DOI: `10.1109/IV51971.2022.9827073`. URL: `https://doi.org/10.1109/IV51971.2022.9827073`.