TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Reinforcement learning in robotic arm position control

Master's thesis

2020 **Bc. Tomáš MERVA**

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Reinforcement learning in robotic arm position control

Master's thesis

Study programme:	Electrical systems
Study specialization:	Electrical and Electronics Engineering
Department:	Department of electrical engineering and mechatronics
Supervisor:	Ing. Peter Girovský, PhD.
Consultant:	Prof. dr. Robert Babuška
	Assoc. Prof. Ing. František Ďurovský, PhD.

2020 Košice **Bc. Tomáš MERVA**

Abstract in English

The purpose of this master's thesis is threefold. The first part provides an introduction to reinforcement learning and the state-of-the-art research of reinforcement learning for problems, which cannot be solved by ordinary methods. The second part of the thesis deals with the implementation of the Soft Actor-Critic algorithm (Haarnoja et al., 2018) in cooperation with Hindsight Experience Replay (Andrychowicz et al., 2017) algorithm, in order to solve robotic test environments with sparse rewards. In the last part the combination of the algorithms is implemented on a robotic arm with the usage of ROS (Robot Operating System).

Key words in English

Reinforcement learning, robotics, ROS

Abstract in Slovak

Predkladaná diplomová práca sa skladá z troch častí. Prvá časť vysvetľuje základy učenia posilňovaním a najnovší výskum v oblasti reinforcement learning pre problémy, ktoré sa nedajú vyriešiť klasickými metódami. Druhá časť diplomovej práce sa zaoberá implementáciou Soft Actor-Critic algoritmu (Haarnoja et al., 2018) v spojení s algoritmom Hindsight Experience Replay (Andrychowicz et al., 2017) na vyriešenie robotických testovacích úloh so zriedkavými odmenami. V poslednej časti diplomovej práce sa aplikuje kombinácia algoritmov na robotické rameno za použitia ROS (Robot Operating System).

Key words in Slovak

Reinforcement learning, robotika, ROS

57687

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Department of Electrical Engineering and Mechatronics

DIPLOMA THESIS ASSIGNMENT

Field of study: **Electrical and Electronics Engineering**

Study programme: **Electrical systems**

Thesis title:

Reinforcement learning in robotic arm position control

Reinforcement learning v riadení polohy robotického ramena

Student: **Bc. Tomáš Merva**

Supervisor: **Ing. Peter Girovský, PhD.**

Supervising department: **Department of Electrical Engineering and Mechatronics**

Consultant: **doc. Ing. František Ďurovský, PhD., prof. Dr. Robert Babuska**

Consultant`s affiliation: **Department of Electrical Engineering and Mechatronics**

Thesis preparation instructions:

1. Study the problem of Reinforcement learning.
2. Design a system for control of the robotic arm.
3. Design robotic arm control using ROS.
4. Implement Reinforcement learning in robotic arm position control.
5. Develop the thesis according to the instructions of the supervisor.

Language of the thesis: English

Thesis submission deadline: 04.05.2020

Assigned on: 31.10.2019

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -02-

12.

prof. Ing. Liberios Vokorokos, PhD.

Dean of the Faculty

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 25. April 2020 .....

Signature

Acknowledgement

Firstly, I would like to thank my supervisor Ing. Peter Girovský, PhD. for his constant support during the last three years in terms of creating amazing research environment thanks to which I could grow as a person and an engineer, and of course for helping me out with this thesis.

My sincere thanks also goes to doc. Ing. František Ďurovský, PhD., who was trying to create as much research possibilities as possible during the last three years for me as well as for my colleagues. Moreover, his continues material and knowledge support was invaluable.

Even–though I was an unknown student from the foreign university, Prof. dr. Robert Babuška was willing to be my consultant of this thesis for which I will have been profoundly grateful. Thanks to him I was capable to discover the magic of reinforcement learning and I have always been motivated to overcome my ideas within this thesis.

A special thanks to everyone from the Institute of Automation, Mechatronics, Robotics and Production Systems for providing the tools, space and freedom to working on my thesis. Furthermore, I would like to thank doc. Ing. Ivan Virgala, PhD. for his tremendous help with ROS.

I cannot forget to thank Ing. František Ďurovský Junior, PhD. for his patience with me regarding ROS and robotics, Mgr. Michal Malý, PhD. for consulting the topic of the thesis and Ing. Viktor Šlapák, PhD. for helping me out with understanding publications.

Last but not the least, I would like to thank my family, because if it weren't for my family, I would not have reached this far.

Preface

The rapid growth of the computers in terms of the computing power has enabled the huge leap in artificial intelligence development. One of the successful examples of the usage of artificial intelligence is the DQN policy (Mnih et al., 2013) that has accomplished not only to be at the same level as human player in playing Atari games, but in a lot of these games it has been even better. For me the most fascinated part of this success is that the policy has learnt to play Atari games without prior human knowledge. This approach can be also used in robotics, with the result that the robots are capable of interacting with the environment that is constantly and for the most part unexpectedly changing. Therefore, the implementation of robots is no more limited to simple repetitive tasks.

I have chosen this topic for my master's thesis with the intention of learning about artificial intelligence, especially reinforcement learning, and deepening my knowledge in robotics using ROS. Regarding ROS this thesis is a continuation of my bachelor's thesis, thanks to which I have got my first experience with this standard of robotics development.

This thesis was conducted in cooperation with the Department of Electrical Engineering and Mechatronics of the Faculty of Electrical Engineering and Informatics and Institute of Automation, Mechatronics, Robotics and Production Systems of Faculty of Mechanical Engineering at the Technical University of Košice.

Content

List of Figures

List of Tables

List of Symbols and Abbreviations

RL	Reinforcement Learning
SAC	Soft Actor-Critic
HER	Hindsight Experience Replay
ROS	Robot Operating System
AI	Artificial Intelligence
tf	Transform Library
IK	Inverse kinematics
DoFs	Degrees of Freedom
fps	Frames per second
DDPG	Deep Deterministic Policy Gradient

1. Introduction

In 2012 the American military research agency DARPA the competition DARPA Robotics Challenge 2015, which held from 2012 to 2015. The purpose was to address and promote innovation in human-supervised robotic technology for disaster-response operations. The primary technical goal was to develop ground robots capable of accomplishing eight complex tasks in difficult environments, such as using hand tools or driving a vehicle [2]. The original requirement was to use fully autonomous robots, although because of the complexity of the challenging tasks, DARPA had to change it to human-supervised robots (teleoperation) [3]. The winning team KAIST from South Korea successfully accomplished all eight tasks in 44 minutes and 28 seconds. However, watching the robot doing each task, such as a door opening, leaves someone wondering if this is the way how people would do it, especially in terms of speed. By the time the team KAIST's robot opened the door, the human would do it around ten times and without any supervision. [4]

The reason for limited applications of robots in the real world is that environment is constantly changing, and it is very difficult to "hard-code" the robot's movement in such environment. Therefore, robots are mainly used to do repetitive and precise tasks in well-known environments. By environment it is meant everything around a robot including objects it interacts with. Another issue is designing robotic control pipeline for autonomous operation. At every stage of the pipeline some kind of mistake can be made, which results in accumulating the error as it goes through other stages and the robot will not do anything useful. Therefore, the whole decision process has to be slowed down in order to avoid that assumptions within each stage are not violated too much. [4]

Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviours. The concept behind reinforcement learning is that it enables a robot to autonomously discover an optimal behaviour through trial-and-error interactions with its environment. Instead of providing a detailed solution to a problem by a supervisor, the robot gets feedback in terms of a numerical value (reward) that measures the one-step performance of the robot. The robot tries to maximize this reward over time. [5] The whole process of learning is inspired by the way how animals or children learn certain tasks. The behaviour that resulted with a small reward will unlikely be repeated, whereas the behaviour with a high reward will. [6]

Reinforcement learning has been famously used to create artificial intelligence for solving strategy games such as Go (Silver et al., 2016) or Dota (OpenAI, 2018). However, using reinforcement learning within robotics differs substantially from most well-studied reinforcement learning benchmark problems. Tasks in robotics are mostly represented with high-dimensional,

continuous states and actions. The observation of the true state is most of the times incomplete and consists of a lot of noise. As a consequence, a robot is not able to know exactly in which state it is. Another issue is that many completely different states seem similar, which results in robot's unreasonable actions. Besides aforementioned problems there are many other challenges such as sparse rewards, model errors or an appropriate learning method. [5]

Model-free reinforcement learning algorithms suffer from two major problems, high sample complexity and convergence properties. With regard to the first challenge, RL methods require an enormous amount of samples of experience. With the higher-dimensional states the RL algorithms need substantially more. The poor sample efficiency is primary caused by on-policy learning, in which the new version of a policy can be created only by using experience collected by the previous version of the policy. This problem can be reduced with off-policy algorithms that learn from and reuse experience obtained by any version of the policy. However, off-policy algorithms have worse properties than on-policy methods in terms of stability and convergence. Typical example of this issue is DDPG algorithm that is good in terms of sample-efficiency but is notoriously problematic to use due to its hyper parameter sensitivity. The recent progress in field of the model-free deep RL algorithms for continuous state and action space is the off-policy algorithm Soft Actor-Critic, which presents sample-efficiency and stability at the same time. This algorithm has successfully solved robotic tasks such as rotating a valve or stacking Lego blocks. [9] [10]

The aim of this work is to test the SAC algorithm extended with the HER algorithm, which is capable to solve the problem with sparse rewards, on custom robotic environments created using ROS. The thesis is divided into three sections. The first section provides all necessarily prerequisites that someone needs in order to understand the second sections, in which the HER and the SAC algorithms are explained. The third section describes creating the custom environments using ROS and summarizes the results of testing SAC + HER in those environments.

2. Reinforcement Learning

Reinforcement learning is one of the fields of machine learning, which can be called as a science of decisions making. In reinforcement learning a learner is not told which actions to take, contrary to the most of areas of machine learning, but instead he must discover which actions yield the most reward by trying them. The learner is denoted by the agent, who interacts with the world, comprising everything outside of the agent, called the environment. After the agent has done an action a_t, he gets from the environment an observation of the environment termed the state s_t and a reward r_t for this particular action. Rewards are numerical values, which the agent tries to maximize over time [7] [8].

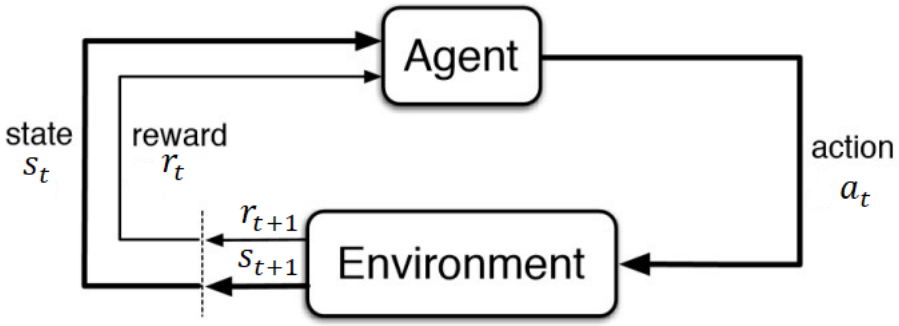

Fig. 1 Scheme of agent-environment interaction [7]

To keep an explanation of reinforcement learning simple, it is considered that the agent and environment interact at discrete time steps t = 0, 1, 2, … At each time step t the agent gets some representation of state s_t and according the state he selects an action a_t. As a consequence of its action, the agent receives a numerical reward r_{t+1} and finds itself in a new state s_{t+1} [7]. The process of this interaction is shown in Fig. 1.

2.1. Policy

The most important part of a human role in making this kind of artificial intelligence is creating a policy denoted as π. A policy π is essentially a map from a state to an action. In psychology it would be called a set of stimulus-response rules. The policy can be a simple function or a lookup table, but also it may involve extensive computation. The policy represents the core of the reinforcement learning agent, because it defines agent's way of behaving at a given time. A reinforcement learning algorithm is used for teaching a policy in a way that it guides changes of the policy's parameters in order to maximize a reward. Policies can be either deterministic or stochastic. A deterministic policy maps a state to an action. It gets an action and the function returns an action to take (1). [7][8]

$$\pi: S \rightarrow A \tag{1}$$

A stochastic policy outputs a probability distribution over actions (2). Instead of being completely sure of taking the action *a*, there is a probability that the different action will be taken. This approach supports exploration of different actions. This thesis deals with stochastic policies. [11]

$$\pi(a|s) => a_t \sim \pi(.|s_t) \tag{2}$$

The two most common kinds of stochastic policies are *categorical policies* and *diagonal Gaussian policies*. [1]

2.1.1. Categorical Policy

A categorical policy is very similar to a classifier in a sense that the neural network is build the same way for the categorical policy as for the classifier. The number of outputs in the final layer of the neural network depends on the number of discrete actions. The neural network outputs logits for each action, followed by the Softmax function in order to convert the logits into probabilities. The action, which will be taken, is chosen by sampling based on the probabilities. With regard to better and easier optimization of a policy it is preferable to use log-likelihood for training purposes. If the last layer of probabilities is denoted as a vector $P_\vartheta(s)$, the log-likelihood for an action *a* can be obtained as follows:

$$\log \pi_\theta(a|s) = \log[P_\theta(s)]_a \tag{3}$$

Therefore, it is convenient to use the Log-Softmax function instead of the classical Softmax function as the activation function for the last layer. In general, it can be said that a neural network outputs the parameters for a categorical distribution. Categorical policies are used in environments with discrete action space [1].

2.1.2. Gaussian Policy

For solving robotic tasks, the RL algorithms have to deal with the large continuous action spaces with an infinite number of actions. Unlike computing probabilities for each action, as it is in case of using categorical policies, Gaussian policies learn statistics of the probability Gaussian (normal) distribution. A Gaussian policy has always a neural network that has a state *s* as an input, and outputs mean actions $\mu_\vartheta(s)$. The other parameter, which defines Gaussian distribution, is a standard deviation $\sigma_\vartheta(s)$. A standard deviation could be represented by a neural network, that maps from states to standard deviation, or by a single vector. In the second case, standard deviations are not outputs from a function, but they are standalone parameters. Either way log-standard deviations are used directly instead of standard deviations. [1][7]

Actions are chosen on the base of the given the mean action $\mu_\vartheta(s)$ and the standard deviation $\sigma_\vartheta(s)$ that is elementwise multiplied with a noise vector *z* (4).

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot \mathrm{z} \tag{4}$$

Since neural networks do not produce probabilities for each action, the log-likelihood of a k-dimensional action *a* is computed as follows:

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right) \tag{5}$$

2.2. Reward and Return

Rewards are one of the most fundamental quantity in RL. A reward r_t is simply a number that indicates how well agent is doing at step *t*. Therefore, the agent's job is to sum up r_t-s and get as much reward as possible. Rewards depend on the current state of the environment, the action just taken and the next state of the environment (6). However, in practice it is simplified to just the current state and the action. [1]

$$r_t = R(s_t, a_t, s_{t+1}) \tag{6}$$

A reward is defined by a reward function, which maps each state (or state-action pair) of the environment to a single number. A reward function basically guides an agent through the whole process of the agent's interaction with environment with the intention of accomplishing a required task. Using a different reward function in the same environment learns an agent to accomplish a different task. An agent gets a reward each timestep, therefore in general the agent's goal is to maximize the return, which can be defined as some specific function of the reward sequence. The simplest function of the return is the sum of the rewards: [1] [7]

$$R_t = \sum_{t=0}^{T} r_t \tag{7}$$

The better function is using the discounted return, which uses a discount factor γ (8). It is the constant with value between 0 and 1. With the bigger discount factor agent cares more about the long-term reward. On the other hand, with the smaller discount factor agent cares more about the immediate (short-term) reward. The undiscounted return is used in case of an episode with a fixed step size, whereas the discounted return is used in continuing tasks. [1] [7]

$$R_t = \sum_{t=0}^{\infty} \gamma^t r_t \tag{8}$$

With the knowledge of what return is, the goal in RL is to select the policy that maximizes expected return when the agent acts according to it [1]. If the expected return is denoted as *J(π)*, then it is expressed as:

$$J(\pi) = E[R(\tau)] \tag{9}$$

where τ is a trajectory, which is a sequence of states and actions in the certain environment:

$$\tau = (s_0, a_0, s_1, a_1, s_2, \dots) \tag{10}$$

With the defined expected return *J(π)*, the goal of the optimization in RL is to find the optimal policy:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{11}$$

2.3. Value functions

The other essential quantity in RL is a value function, which tells an agent how good it is to be in a certain state or how good it is to perform a given action in a certain state. A value function estimates how much total reward an agent should expect in the future. The future rewards depend on agent's actions, therefore a value function is defined with respect to a given policy. A large majority of RL algorithms use value functions. There are four main value functions: [1] [8]

1) The *state-value function* is the expected return when starting in the state *s* and following policy π thereafter:

$$V^{\pi}(s) = E_{\pi}[R_t | s_t = s] \tag{12}$$

2) The *action-value function* is the expected return when starting in the state *s*, taking the action *a* and following policy π thereafter:

$$Q^{\pi}(s, a) = E_{\pi}[R_t | s_t = s, a_t = a] \tag{13}$$

3) The *optimal state-value function* is the expected return when starting in the state *s* and following the optimal policy π thereafter:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{14}$$

4) The *optimal action-value function* is the expected return when starting in the state *s*, taking the action *a* and following the optimal policy π thereafter:

$$Q^*(s) = \max_{\pi} Q^{\pi}(s, a) \tag{15}$$

In Deep RL neural networks are used as approximators of optimal value functions in a way that a state or a state-action pair inputs into the neural network, which outputs the value of the state *s* or value of taking the action *a* in the state *s*. As the policy converges to the optimal policy, so the value function network converges to the optimal value function. [8]

2.3.1. Bellman Equations

One of the most fundamental relationships in RL is called Bellman equation. The equation decomposes state-value functions into two parts, the immediate reward r_t and the discounted value of a successor state $\gamma V^{\pi}(s_{t+1})$, in order to express a relationship between the value of a state and the values of its successor states. In case of state-value functions, the equation is called the Bellman equation for V^{π}. [7] [8]

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}[R_t|s_t = s] \\ &= E_{\pi}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots |s_t = s] \\ &= E_{\pi}[r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \cdots)|s_t = s] \\ &= E_{\pi}[r_t + \gamma R_{t+1}|s_t = s] \\ &= E_{\pi}[r_t + \gamma V^{\pi}(s_{t+1})|s_t = s] \end{aligned} \tag{16}$$

Accordingly, the same principle can be applied to the action-value function $Q^{\pi}(s,a)$:

$$Q^{\pi}(s,a) = E_{\pi}[r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \tag{17}$$

It is important to realize the connection between a state-value function and an action-value function. One value function can be expressed by the other one. In the Fig. 2 each empty circle represents a state and the solid circles represent possible actions. [1] [7]

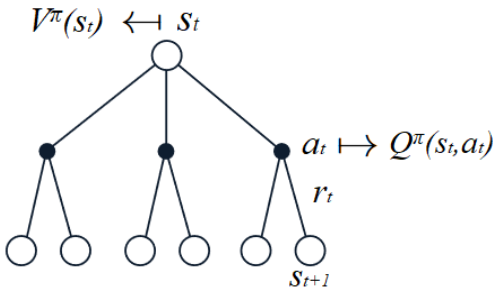

Fig. 2 Backup diagram for $V^{\pi}(s)$ [7]

Assume that the agent is in the state s_t and he can choose from three possible actions based on the probabilities defined by the policy. For each of the actions the agent might take, there is the Q-value (action-value function) that tells the agent how good it is to take the certain action from the state. In order to figure the value of the state-value function for the state s_t, it is necessary to look ahead one step to the Q-values of the actions and average them as follows:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s)\, Q^{\pi}(s,a), \tag{18}$$

where A is action space (all possible actions) and $\pi(a|s)$ is probability of taking an action in a given state defined by a policy. To sum up value functions, a state-value function tells an agent how good

it is to be in a particular state, whereas an action-value function tells the agent how good it is to take a particular action from a given state. [7][8]

2.4. Actor-Critic Taxonomy

2.4.1. Critic-only Methods

The best example of critic-only methods is the Q-learning algorithm. Instead of using an explicit function for the policy, the Q-learning algorithm is based on using an action-value function to determine a particular action from a given state. The simplest and original version of the Q-learning algorithm is to use a lookup table. The columns are all possible actions, whereas the rows are all possible states. The value of each cell is the expected reward for the given state and action and the agent chooses the action with the biggest value among the all possible actions (19). The modern version of the Q-learning algorithm uses a neural network as the Q-value approximator. The outputs from the neural network represent Q-values of each possible action. This form of the algorithm is called Deep Q-learning. [1] [6] [11]

$$a(s) = \arg\max_{a} Q(s, a) \tag{19}$$

2.4.2. Actor-only Methods

On the other hand, actor-only methods work with some function for the policy and do not use any form of a value function. The most common actor-only methods are policy gradient methods that optimize an objective function $J(\pi_\vartheta)$. In terms of RL the cost function is the expected return (9). Policy gradient algorithms optimize policy's parameters θ by gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta), \tag{20}$$

where $\nabla_\theta J(\pi_\theta)$ is called the policy gradient and its simplest form can be estimated as follows:

$$\hat{g} = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{21}$$

The idea behind using policy gradient is to increase or decrease the probability of an action in order to find a parameter θ that maximizes the expected return. The log-likelihood for an action a is multiplied by the return for a whole episode. If the episode was successful, the probability of taking the same action in a given state is increased. However, multiplying the log-likelihood of an action with the return creates a big problem. If the return is high, all actions that the agent took are considered as good, even though some were really bad. In the following figure there is shown that even the bad action will be evaluated as a good one, because averaging all rewards from each action will result in the high return. [1] [6] [11]

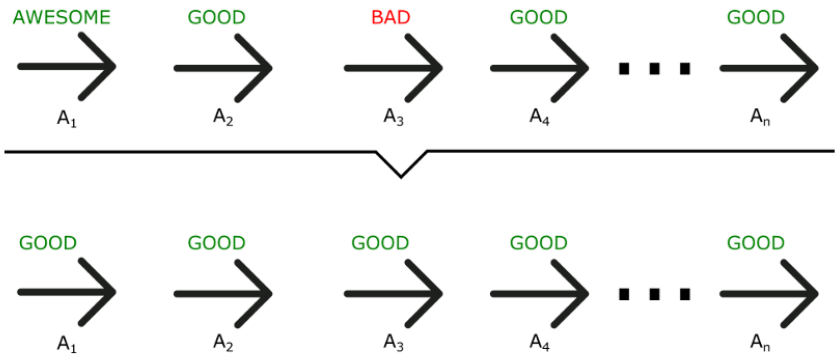

Fig. 3 Disadvantage of simplest policy gradient [11]

2.4.3. Actor-Critic Methods

In terms of actor-critic algorithms, the learning agent consists of two separate entities. The policy structure that is responsible for generating actions is called the actor. The second entity is the estimated value function known as the critic. The concept of an actor-critic algorithm is very similar to a scenario of two friends playing a video game. At the beginning of playing the unknown game, the player is randomly interacting in the video game environment, whereas the other friend is carefully observing the player's action and providing feedback. As a consequence, the player (actor) is getting better in choosing right actions but the critic is also improving on providing better feedback. With regard to modern RL, having the actor-critic structure means having two separate neural networks (one for the actor, the other one for the critic), which must be optimized separately. [1] [6] [11]

For the purpose of better explanation, the policy gradient can be expressed in a general form:

$$\nabla_\theta J(\pi_\theta) = E\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta\,(a_t|s_t)\psi_t\right] \tag{22}$$

where ψ_t can be any of the following:

1. $R(\tau)$
2. $\sum_{t=t}^{T} r_t$
3. $\sum_{t=t}^{T}\big(r_t - b(s_t)\big)$
4. $Q^\pi(s_t, a_t)$
5. $Q^\pi(s_t, a_t) - V^\pi(s_t)$
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$

Usually, a state-value function is chosen as the critic. Each timestep the critic evaluates whether the new state is better or worse than the previous one. This evaluation is called the TD error and it is expressed as $\psi_t = r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$. The following figure depicts the actor-critic structure in case of using the TD error as a guidance for the actor. Naturally, the aforementioned evaluation can have a different form than the TD error. [6] [16]

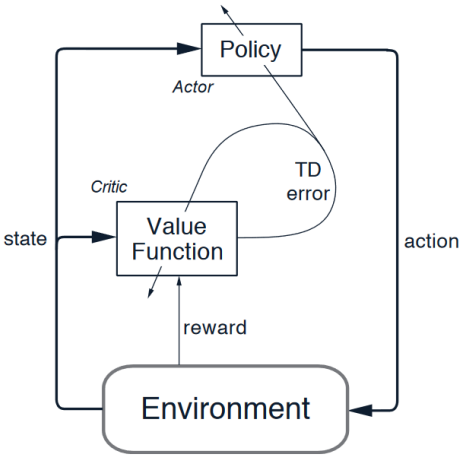

Fig. 4 Actor-critic structure [6]

3. Hindsight Experience Replay

One of the challenges within solving robotic tasks is dealing with sparse rewards. The state space of robotic environments is hard to explore because of its large size. As a consequence, an agent receives the reward $r_t = -1$ almost every timestep, which results that a standard RL algorithm is bound to fail. Despite the problems with sparse rewards, using dense rewards is much more inconvenient due to the need to design the right reward function that is capable of reflecting the task precisely but is also carefully shaped in order to guide the policy optimization. Therefore, there is an effort to devise new algorithms that allow sample-efficient learning from sparse rewards. The Hindsight Experience Replay algorithm does just that and is easy to implement. [17] [18]

Assume that a human is to slide a puck across the table in order to get it to the goal destination. The first attempt did not accomplish the desired goal, but it achieved a different one. Even though the first attempt was not successful, the person gains experience about the required strength and angle for accomplishing the achieved goal. The gained experience will be used in the next attempts. With each unsuccessful attempt, the person is closer and closer to accomplish the task. This process of learning, learning from mistakes, inspired OpenAI to devise HER. After an unsuccessful episode, the achieved goal substitutes the desired one and the agent can obtain a learning signal since it has achieved some goal. Repeating this process will result in successfully accomplishing arbitrary goals as well as the desired goals. [17] [18]

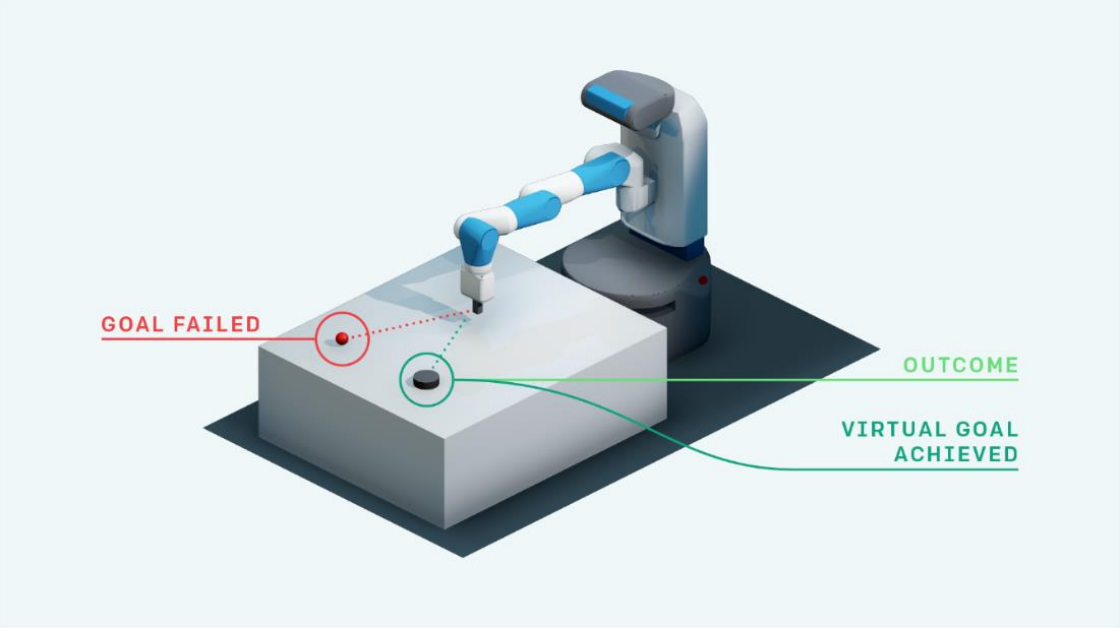

Fig. 5 Idea of HER [18]

The next advantage of HER is that it enables an agent to be flexible in terms of accomplishing different goals in the same environment. This kind of environment is called the goal-based or goal-oriented environment. Each possible state of the environment can be considered as a separate goal. The goal can also specify only certain properties of the environment, for example just one coordinate. In order to want to learn only one specific goal, it is still recommended to learn multiple goals since HER learns faster using multiple goals. In the following figure is shown that using only the DDPG algorithm is unable to solve the pushing task (a robotic arm was to push a block to a goal position) without HER. [17] [18]

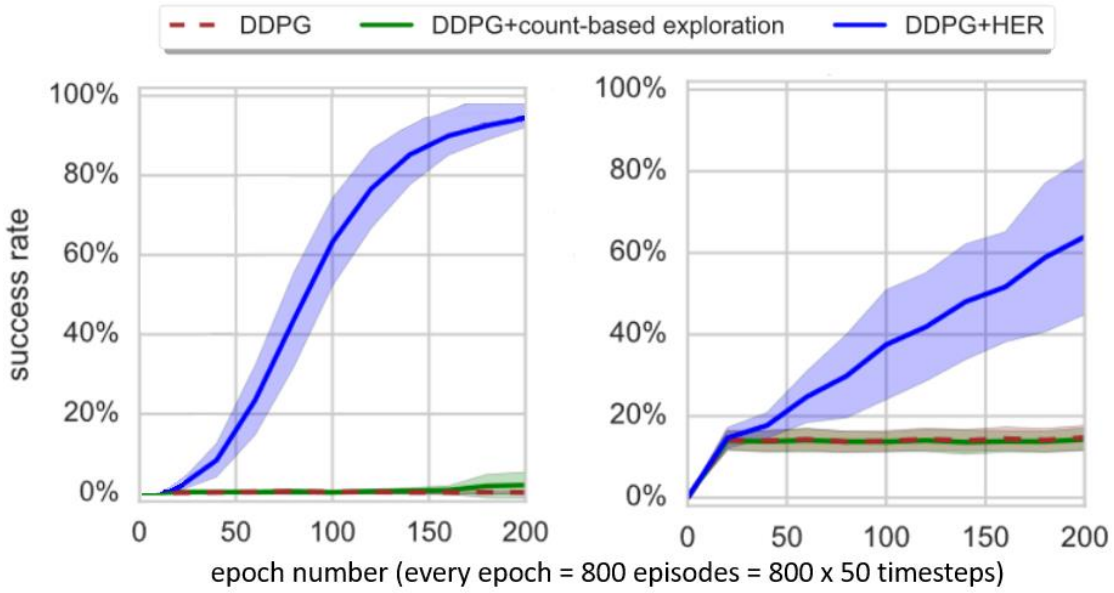

Fig. 6 Comparison of using single goal with using multiple goals [17]

(Left: multiple goals, Right: single goal)

In case of using a neural network as a policy, the input consists of a given state but also of a desired goal.

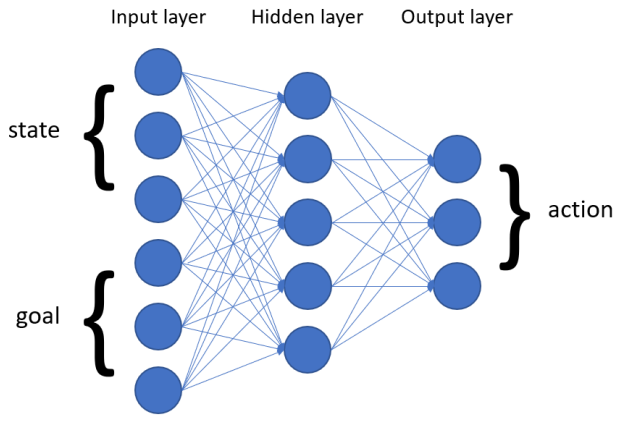

Fig. 7 Neural network with additional goal

$$\pi(a|s) \rightarrow \pi(a|s,g) \tag{23}$$

The replay buffer within HER consists of transitions of each timestep. Each transition is a list of four tuples $(s_t\|g, a_t, r_t, s_t'\|g)$, where $\|$ denotes concatenation. Assume the example in which the agent has failed a given task after four timesteps (t_0, t_1, t_2, t_3) and has not achieved a given goal g. However, agent has ended up in the certain state s_4, which can be denoted as $g' = s_4$ and it will serve as the arbitrary goal. So far, the replay buffer contains four transitions:

Tab. 1 Original transition in replay buffer

timestep	$s_t\|g$	a_t	r_t	$s_{t+1}\|g$
t_0	$s_0\|g$	a_0	$r_0 = -1$	$s_1\|g$
t_1	$s_1\|g$	a_1	$r_1 = -1$	$s_2\|g$
t_2	$s_2\|g$	a_2	$r_2 = -1$	$s_3\|g$
t_3	$s_3\|g$	a_3	$r_3 = -1$	$s_4\|g$

For HER each transition has to be stored twice, although with a different goal. After saving the original transitions, the second transition is created by copying the first one but with an arbitrary goal g' instead of the original goal g. As a result of the substitution, the agent obtains a learning signal.

Tab. 2 Additional transition in replay buffer

timestep	$s_t\|g'$	a_t	r_t	$s_{t+1}\|g'$
t_0	$s_0\|g'$	a_0	$r_0 = -1$	$s_1\|g'$
t_1	$s_1\|g'$	a_1	$r_1 = -1$	$s_2\|g'$
t_2	$s_2\|g'$	a_2	$r_2 = -1$	$s_3\|g'$
t_3	$s_3\|g'$	a_3	$r_3 = 0$	$s_4\|g'$

HER can be implemented using four different strategies. The aforementioned process is called the *final* strategy, because the additional goals correspond to the final state of the environment. The other three strategies are: *future*, *episode* and *random*. For the purposes of this thesis, an episode is randomly divided into four parts, which results in four arbitrary goals and consequently the agent has four new learning signals.

Tab. 3 Pseudocode of HER [17]

Initialize a policy π, learning algorithm and empty buffer *D*
for episode = 1,…,*M* **do**
  Sample a goal *g* and an initial state s_0
  for t = 0,…,*T*-1 **do**
    Sample an action a_t using the policy π: $a_t \leftarrow \pi(s_t||g)$
    Execute the action a_t and observe a new state s_{t+1}
  end for
  Sample a set of additional goals for replay $G := S$ from the current episode
  for t = 0,…,*T*-1 **do**
    $r_t = r_f(s_t, a_t, g)$, where r_f is a reward function
    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in *D*
    for $g' \in G$ **do**
      $r' = r_f(s_t, a_t, g')$
      Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in *D*
    end for
  for t = 1,…,*N* **do**
    Sample a minibatch *B* from the replay buffer *D*
    Perform optimization using the learning algorithm and the minibatch *B*
  end for
end for

HER

4. Soft Actor-Critic

Before explaining the Soft Actor-Critic algorithm it is necessarily to introduce few tricks and concepts that are essential parts of the algorithm.

4.1. Maximum Entropy RL

In 1948 Claude Shannon founded the "Information theory" that was aimed for reliable and efficient transmitting a message from a sender to a recipient. Based on this concept the term entropy has found an application in informatics. In general, entropy is the average of amount of information that someone gets from one sample drawn from the given probability distribution *p* (16). It says how unpredictable the probability distribution *p* is or how random a random variable is. The easiest example of what entropy represents is a tossing a coin. There is a 50/50 chance of either outcome, so it has high entropy. If a coin is biased so that it always comes up the head, then entropy is low. [1] [12] [13]

$$H(p) = -\sum_{i} p_i \log_2(p_i) \tag{24}$$

As it has already been mentioned, the standard goal of an agent is to maximize the expected sum of reward:

$$\pi^* = \arg\max_{\pi} J(\pi) = \arg\max_{\pi} \sum_{t=0} E[r(s_t|a_t)] \tag{25}$$

Instead of maximizing only the expected return, maximum entropy RL optimizes policies also with the expected entropy of the policy. Therefore, maximum entropy adds the expect entropy of the policy into the standard RL objective as follows:

$$J(\pi) = \sum_{t=0} E\big[r(s_t, a_t) + \alpha . H\big(\pi(.|s_t)\big)\big], \tag{26}$$

where the parameter α is called the temperature parameter, which determines the stochasticity of the optimal policy. The higher coefficient α supports more exploration of the environment, whereas in the limit $\alpha \to 0$ the standard maximum expected return objective is recovered. In order to ensure that the sum of expected returns and entropies is finite, it is common practice to use a discount factor *γ*. Consequently, the 18th equation is modified as follows:

$$J(\pi) = E \sum_{t=0} \gamma^t \big[r(s_t, a_t) + \alpha . H\big(\pi(.|s_t)\big)\big] \tag{27}$$

This form of the objective function has three main advantages. The first benefit is that the policy is forced to explore widely and unpromising areas are ignored. The second advantage is related to

the first one. As the result of improved exploration, the learning speed is improved over methods using the standard objective. The last but not least advantage is that in cases where multiple actions seem to be equally promising, the probability mass of those action would be equal. From the 19th equation the new state-value function $V^{\pi}(s)$ can be derived so it also includes the entropy bonuses:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) + \alpha H\big(\pi(.|s_t)\big)\right) \middle| s_0 = s\right] \tag{28}$$

Accordingly, the new action-value function $Q^{\pi}(s,a)$ can be expressed as follows:

$$Q^{\pi}(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) + \alpha \sum_{t=1}^{\infty} \gamma^t H\big(\pi(.|s_t)\big) \middle| s_0 = s, a_0 = a\right] \tag{29}$$

With these definitions, the soft state-value function can be obtained as:

$$V^{\pi}(s) = E\big[Q^{\pi}(s,a) + \alpha H\big(\pi(.|s)\big)\big] \tag{30}$$

and by using the Bellman equation for the action-value function, the soft action-value function is expressed as: [1] [12] [13]

$$\begin{aligned} Q^{\pi}(s,a) &= E\left[r(s_t, a_t) + \gamma\left(Q^{\pi}(s_{t+1}, a_{t+1}) + \alpha H\big(\pi(.|s_{t+1})\big)\right)\right] \\ &= E[r(s_t, a_t) + \gamma V^{\pi}(s_{t+1})] \end{aligned} \tag{31}$$

4.2. Target Network

In order to understand SAC, it is also necessary to understand the concept of using target networks in RL. The aforementioned tabular Q-learning algorithm updates a Q-function using the Bellman equation:

$$Q_{k+1}(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \alpha\left[r_t + \gamma \max_{a_{t+1}} Q_k(s_{t+1}, a_{t+1}) - Q_k(s,a)\right], \tag{32}$$

where $r_t + \gamma \max_{a_{t+1}} Q_k(s_{t+1}, a_{t+1})$ is called $target(s_{t+1})$ so the previous equation can be expressed as follows:

$$Q_{k+1}(s_t, a_t) \leftarrow Q_k(s_t, a_t) + \alpha[target(s_{t+1}) - Q_k(s,a)] \tag{33}$$

The other version of this algorithm is called Approximate Q-learning. Instead of using a table, a parametrized Q function $Q_{\theta}(s,a)$ is used in the form of a linear function or a complicated neural network. Update of a parametrized Q-function is defined by:

$$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_{\theta_k}\left[\frac{1}{2}\Big(Q_{\theta_k}(s_t, a_t) - target(s_{t+1})\Big)^2\right] \tag{34}$$

Using neural networks with Q-learning requires some tricks. The objective of the previous equation is to get the Q-value to approximate to the target Q-value. When weights of the network are updated, the outputted Q-values will be changed but so will the target Q-values since they are computed using the same weights. As a consequence, the Q-values with each iteration are moving closer to the target Q-values but the target Q-values are also moving to the same direction, so they are further and further from the Q-values as a result. This results in chasing a nonstationary target, which makes the optimization unstable. The solution is to create a completely separate network called target network that outputs target Q-values. The target network is a clone of original network, although its weights are updated every certain number of timesteps instead of every iteration. The parameters of the target network are denoted θ_{targ}. In case of algorithms based on Deep Q-learning Networks (DQN), the target network is cloned of the main network every fixed number timesteps. However, in case of the SAC algorithm, the target network is updated once per main network update by using *polyak averaging*:

$$\theta_{targ} = \rho\theta_{targ} + (1-\rho)\theta, \tag{35}$$

where ρ is a hyperparameter between 0 and 1. [1] [21] [22]

4.3. Clipped Double Q-learning

The disadvantage of deep Q-learning is overestimation bias that is caused by maximization of a noisy value estimate. At the beginning of learning, the action-value function Q^{π} is not the optimal action-value function Q^{*}. Therefore the Q^{π} function is not precise, with the result that noisy estimates are unavoidable. An imprecise estimate within each update accumulates the error that causes bad states to be considered as good states. Consequently, the policy update is not done in the best possible way, which results in worse learning performance. The problem of overestimation bias also persists in an actor-critic setting. Therefore, the Twin Delayed Deep Deterministic policy gradient algorithm (TD3) has presented the solution to use two separate action-value functions, with the intention of having two independent estimates that can be used to make unbiased estimate. However, the critics in this form are not entirely independent and it can happen that in certain areas of the state space an action-value can be overestimated. This can be solved by clipping the values estimates in terms of taking the minimum between the two action-values. This is called the Clipped Double Q-learning algorithm. [19]

These two action-value approximators are updated using a mean-squared Bellman error (MSBE) function. The Bellman equation describing the optimal action-value function $Q^{*}(s,a)$ is a starting point for updating the two action-value approximators. Suppose the approximators are neural networks $Q_{\phi 1}, Q_{\phi 2}$ and there is an experience buffer *D* with transitions $(s_t, a_t, r_t, s_{t+1}, d_t)$,

where d_t indicates whether the state s_{t+1} is the terminal state or not. The MSBE function, which expresses how close an action-value approximator is to satisfy the Bellman equation for the optimal action-value function, is defined as follows: [1] [19]

$$L(\phi_1, D) = E_{(s_t,a_t,r_t,s_{t+1},d_t)\in D}\left[\left(Q_{\phi 1}(s_t,a_t) - \left(r_t + \gamma(1-d_t)\max_{a_{t+1}} Q_{\phi 1}(s_{t+1},a_{t+1})\right)\right)^2\right] \quad (36)$$

$$L(\phi_2, D) = E_{(s_t,a_t,r_t,s_{t+1},d_t)\in D}\left[\left(Q_{\phi 2}(s_t,a_t) - \left(r_t + \gamma(1-d_t)\max_{a_{t+1}} Q_{\phi 2}(s_{t+1},a_{t+1})\right)\right)^2\right] \quad (37)$$

4.4. SAC Algorithm

The implementation within this master thesis uses a state-value approximator V_ψ and its target network $V_{\psi_{targ}}$. The objective for the action-value approximators can be modified by using the target state-value approximator:

$$L(\phi_i, D) = E_{(s_t,a_t,r_t,s_{t+1},d_t)\in D}\left[\left(Q_{\phi i}(s_t,a_t) - \left(r_t + \gamma(1-d_t)V_{\psi_{targ}}(s_{t+1})\right)\right)^2\right], \quad (38)$$

where *i* = 1, 2. The target state-value network $V_{\psi_{targ}}$ is obtained by polyak averaging the state-value network parameters over the course of training:

$$\psi_{targ} = \rho\psi_{targ} + (1-\rho)\psi, \quad (39)$$

whereas the learning rule for the state-value network V_ψ comes out of the equation of a soft state-value function.

$$\begin{aligned} V^\pi(s) &= E\left[Q^\pi(s,a) + \alpha H\left(\pi(.|s)\right)\right] \\ &\approx Q^\pi(s,\tilde{a}) - \alpha \log \pi(\tilde{a}(s)|s), \end{aligned} \quad (40)$$

where $\tilde{a}\sim\pi(.|s)$. The loss function for the state-value function is minimized using a mean-squared error based on the approximation defined in the previous equation. Since the implementation of SAC uses two action-value functions, the same concept of the Clipped Double Q-learning algorithm is used. Therefore, the loss function takes the minimum Q-value between the two approximators.

$$L(\psi, B) = E_{\substack{s\in D \\ \tilde{a}\sim\pi_\theta}}\left[\left(V_\psi(s) - \left(\min_{i=1,2} Q_{\phi i}(s,\tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s)\right)\right)^2\right] \quad (41)$$

It is important to realize that in terms of updating the state-value function, actions from the experience buffer are not used. Actions $\tilde{a}$ are sampled fresh from the current version of the policy based on given states from the experience buffer.

The last step of SAC is to update the policy network. Optimizing the policy requires the reparameterization trick, in which an action $\tilde{a}$ is drawn by computing a deterministic function. Instead of using a Gaussian policy from the second chapter, it is used a *Squashed Gaussian policy*. The difference is that the original form of a Gaussian policy is within the hyperbolic tangent function. This ensures that actions are in finite range <-1; 1>. [1] [10]

$$\tilde{a}(s,z) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \mathrm{z}) \tag{42}$$

In terms of SAC the goal of the policy is to maximize the expected future return, as it is the standard RL objective, but also expected future entropy. From this statement, it can be concluded that the objective of SAC is to maximize the soft state-value function $V^\pi(s)$. Therefore, the objective can be defined as follows:

$$\max_\theta E_{\substack{s\in D \\ z\sim N}} \left[\min_{i=1,2} Q_{\phi i}\left(s, \tilde{a}_\theta(s,z)\right) - \alpha \log \pi_\theta(\tilde{a}_\theta(s,z)|s) \right] \tag{43}$$

As a result of maximizing trade-off between of reward and entropy, entropy must be unique to state. Therefore, instead of using the constant standard deviation σ_θ, the standard deviation is output from a neural network. However, it has to be bounded to a certain interval, because at the beginning of the training, the neural network can output large values, which can break the learning. [1] [10]

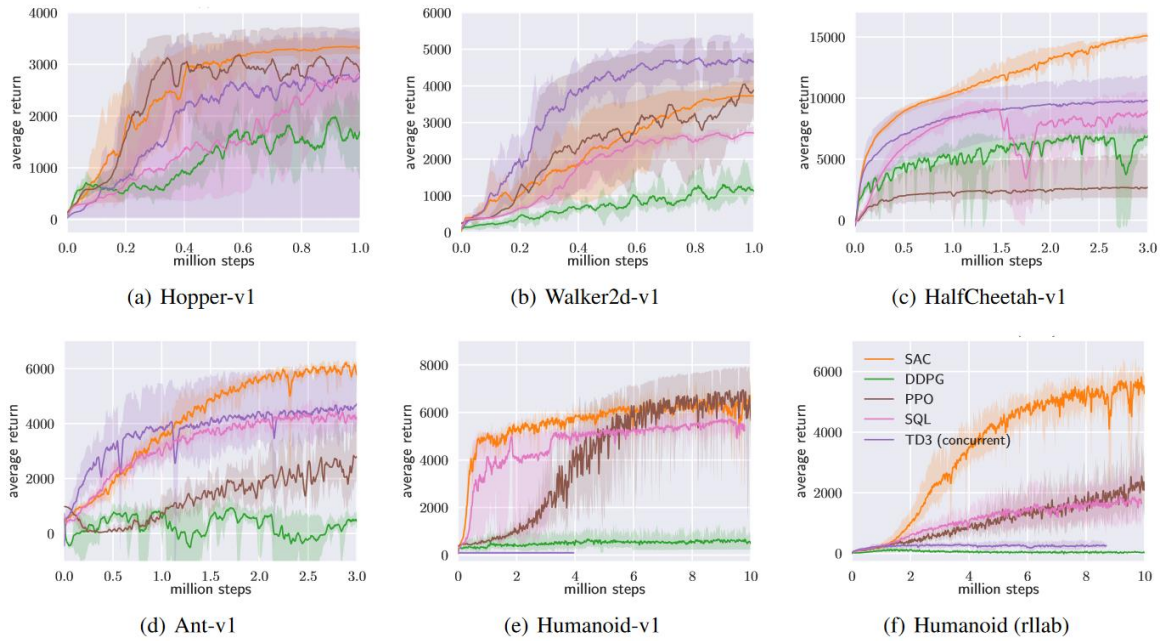

Fig. 8 Comparison of training curves of different algorithms with SAC (yellow) on continuous control benchmarks [10]

4.4.1. Automating Entropy Adjustment

The downside of using SAC is its brittleness to the temperature parameter α. In maximum entropy RL scaling of reward function has negative effect and it has to be compensated by the suitable temperature parameter α. However, the temperature parameter is another hyperparameter that is chosen by a programmer and its wrong value can drastically degrade learning performance. The suitable value of the temperature parameter depends on a given task and environment. Moreover, the high value of the temperature parameter $\alpha \to 1$ forces an agent to explore more, whereas $\alpha \to 0$ corresponds to more exploitation. At the beginning of the training it is preferable, or better said required, to explore the action and state space more but as the agent is getting better, exploration should be less supported. As a consequence, the author of SAC has extended the original algorithm with a constrained formulation that automatically tunes the temperature parameter. This modification accelerates the learning and most importantly provides better stability of learning performance. [23]

The temperature parameter α can be updated using gradient of the following objective function:

$$J(\alpha) = E\left[-\alpha_t \log \pi_{\theta_t}(a_t|s_t;\alpha_t) - \alpha_t \bar{H}\right], \tag{44}$$

where $\bar{H}$ is desired minimum expected entropy. The update of the temperature parameter is performed each gradient step. In the following figure there is shown the comparison of using the constant value of the temperature hyperparameter with using the adjustable one. The experiments were performed on the robotic push task with binary sparse rewards. [23]

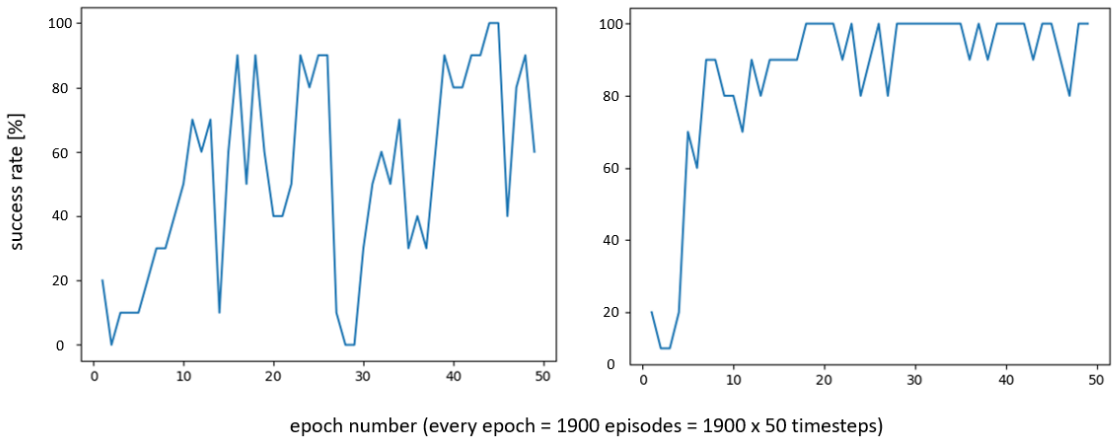

Fig. 9 Comparison of using different approaches within temperature parameter α

(Left: constant α, Right: automatically tuned α)

Tab. 4 Pseudocode of SAC [1]

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, V-function parameters ψ, empty replay buffer *D*

Set targets parameters equal to main parameters $\psi_{targ} \leftarrow \psi$

repeat

Observe state *s* and select action $\alpha \sim \pi_\theta(.|s)$

Execute α in the environment

Observe next state *s'*, reward *r* and done signal to indicate whether *s'* is terminal

Store (s, a, r, s', d) in replay buffer *D*

If s' is terminal, reset environment state

if it is time to update **then**

for *j* in range(however many updates) **do**

Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from *D*

Compute targets for Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{targ}}(s')$$

$$y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \qquad \tilde{a} \sim \pi_\theta(.|s)$$

Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \Big(Q_{\phi,i}(s, a) - y_q(r, s', d) \Big)^2 \qquad \text{for } i = 1,2$$

Update V-function by one step of gradient descent using

$$\nabla_\psi \frac{1}{|B|} \sum_{s \in B} \Big(V_\psi(s) - y_v(s) \Big)^2$$

Adjust temperature parameter α with learning rate λ using

$$\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$$

Update policy by one step of gradient descent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(Q_{\phi,1}\big(s, \tilde{a}_\theta(s)\big) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|s\right) \Big),$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(.|s)$ that is differentiable with θ via the reparametrization trick

Update target value network with

$$\psi_{targ} \leftarrow \rho \psi_{targ} + (1 - \rho)\psi$$

end for

end if

until convergence

5. Implementation of RL Within ROS

5.1. ROS

The essence of the Robot Operating System (ROS) consists of a collection of tools, libraries and drivers that makes the development of robotic applications faster. The main advantage is its open-source ideology that has created the international community of robot researchers and engineers all around the world. This results in even faster progress in the field of robotics. Last years, ROS has also found its utilization within industrial applications and has become the most used platform for robotic development. Therefore, it is a logical choice to use ROS with regard to testing SAC+HER within robotics. For investigating the SAC+HER algorithms, in this thesis it was decided to employ a KUKA Agilus KR 6 R900 CR robot that is fully supported within ROS in terms of drivers and models.

The PC with ROS Melodic represents the main control system for controlling the robotic arm. The PC receives the actual state of the joints of the robot from the KUKA KR C4 Compact controller. The PC interprets those signals and sends accordingly desired goals for each joint of the robot to the KR C4 controller. Data between the PC and the KR C4 controller are transmitted via the TCP/IP protocol and the minimum cycle time takes 2 *ms*, depending on the data volume and the programming method.

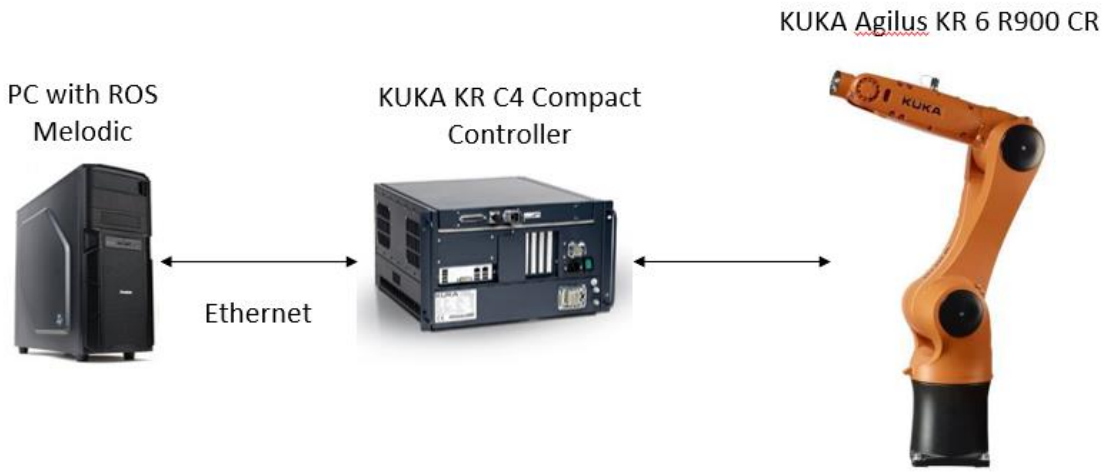

Fig. 10 System Overview

5.2. KUKA Agilus

KUKA Agilus is a compact 6DoF and approximately 52 kg weighing small robot designed for high working speeds. Its possibilities of installation to different positions allow a wide range of applications. The maximum reach of Agilus KR 6 R900 CR is 901.5 mm with a rated payload of 3 kg.

The maximum payload is 6 kg, although in this case the dynamic performance of the robot is not optimized. [24]

Tab. 5 Axis data [24]

Joint	Motion range [°]	Minimum [°]	Maximum [°]
A1	340	-170	170
A2	235	-190	45
A3	276	-120	156
A4	370	-185	185
A5	240	-120	120
A6	700	-350	350

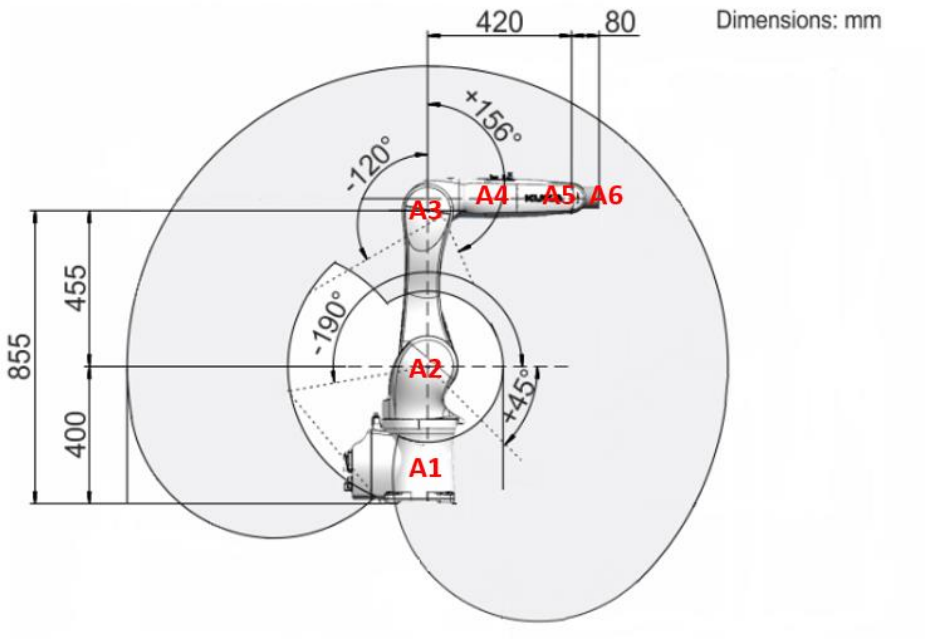

Fig. 11 Workspace graphic [24]

At the end of the joint A6 there is a gripper SMC MHS3-40D, which is a three-finger parallel style pneumatic gripper. The three-finger design enables applications that require gripping small round objects. Even though this thesis does not deal with gripping, it is necessarily to consider its presence within kinematics and motion control. However, this gripper is replaced in the KUKA Push task, because its shape caused throwing a block away from the robot workspace. Therefore, the simple model of cuboid is used that functions as a one finger-type gripper. This exchange solved the problem with undesirable movement of the block.

5.2.1. Robot Model

In order to be able to use the ROS capabilities, especially kinematics and motion planning libraries, it is necessarily to create a precise model of the robotic arm. Fortunately, the models of almost all KUKA robots have already been created and are available as a part of the ROS-Industrial project. So in the URDF model of the robot there had to be added the aforementioned gripper. The easiest and most reliable procedure was to find a STL file at the manufacturer's website. Consequently, physics properties (such as inertia, position limits, maximum joint effort, …) had to be appended to the URDF file. To be able to use the Gazebo simulator, it was necessarily to also add the Gazebo plugin and to specify a transmission for every non-fixed joint so Gazebo would know what to do with a joint. After these steps the model was prepared for being compatible with MoveIt and could be successfully shown in Rviz and Gazebo as follows:

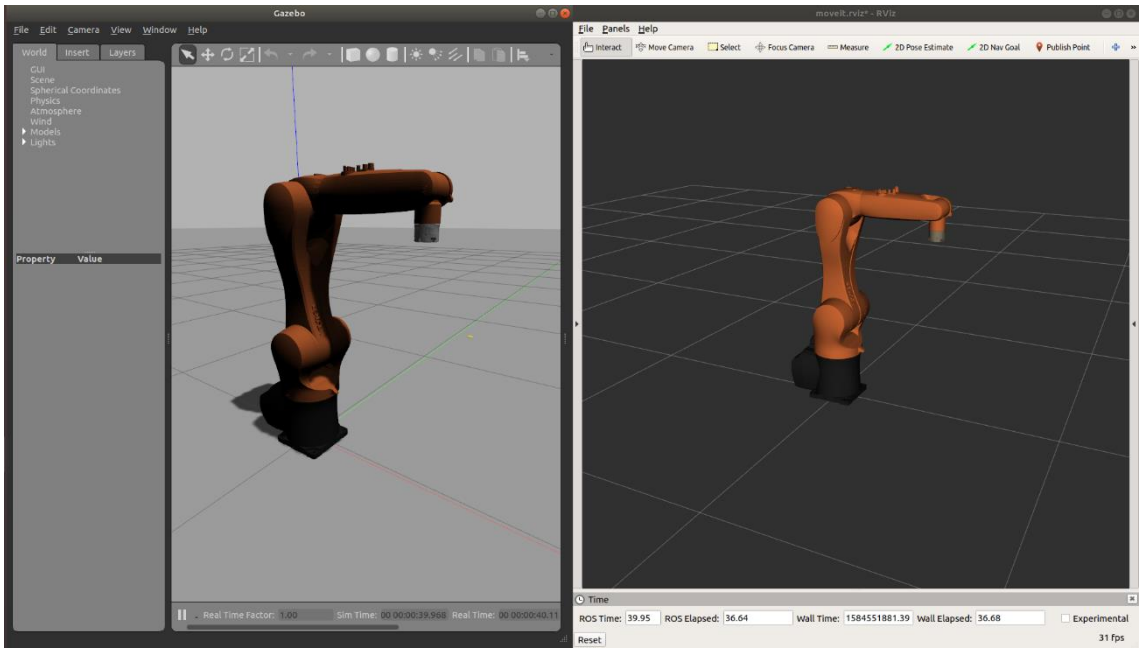

Fig. 12 KUKA Agilus model

5.2.2. Robot Kinematics

ROS provides an easy-to-use platform called MoveIt for developing robotic manipulation applications that is capable to solve forward and inverse kinematics (IK) as well as to plan trajectories. Furthermore, it is compatible with other features of ROS. The default option for the IK solver is the OROCOS Kinematics and Dynamics Library solver (KDL) that use numerical IK approach: using the Newton method or similar to iterate until the solution is found. However, the KDL solver suffers from the following issues: [14]

1) frequent convergence failures for robots with joint limits
2) no actions taken when the search becomes "stuck" in local minima

3) problem handling tolerances for Cartesian pose and the IK solver itself

With regard to the thesis, this downside had extremely negative impact on learning process. With the usage SAC+HER, the policy learnt to output values to which KDL could not find a solution. Therefore, there was an effort to find a better solver that would achieve required behaviour. The solver that satisfies all the requirements is the TRACK-IK solver developed by TRACLabs. The TRACK-IK solver uses a simple extension of the original KDL solver that mitigates the problem with local minima. In additional, the TRACK-IK solver concurrently computes the second IK implementation, which uses quasi-Newton methods that handle joint limits better. By default setting, the TRACK-IK solver provides the faster solution computed by one of these methods. In case of being stuck, the solver automatically restarts itself unlike the KDL solver. From the practical view, the TRACK-IK has better performance in terms of success in finding solutions but also in terms of required time for finding a solution. In the following table it is shown the comparison of KDL and TRACK-IK using different types of robotic manipulators. [14]

Tab. 6 Comparison of KDL and TRACK-IK [14]

Manipulator	DoFs	KDL solve rate	KDL avg. time	TRACK-IK solve rate	TRACK-IK avg. time
ABB IRB120	6	39,41 %	3,08 ms	99,96 %	0,24 ms
Baxter arm	7	61,43 %	2,15 ms	99,83 %	0,37 ms
Fetch arm	7	93,28 %	0,65 ms	99,98 %	0,24 ms
PR2 arm	7	84,70 %	1,26 ms	99,96 %	0,31 ms
UR5	6	16,52 %	4,21 ms	99,17 %	0,37 ms
UR10	6	14,90 %	4,29 ms	99,33 %	0,36 ms

5.3. Basic Motion Control Architecture

To ensure that the position of each joint is correct during the whole trajectory execution, the *ros_control* package is used. The *ros_cotrol* package provides real-time robot controllers that take an actual and desired state of a joint and control the output sent to an actuator by means of a generic control loop feedback mechanism (usually PID). The output can be in form of effort, velocity or position depending on the used controller. The *ros_control* package also provides the high-level controller called Joint Trajectory Controller. MoveIt generates a trajectory, which is basically a set of waypoints consisting of positions, velocities and efforts for joints. Consequently, the role of Joint Trajectory Controller is to ensure that these waypoints are executed at specific time instants.

With regard to the thesis, Joint Trajectory Controller is set to output only desired positions for each joint to the KUKA controller. The superior control system determines the [X, Y, Z] coordinates of an end-effector together with its orientation described by the quaternion [x, y, z, w] and inputs them to the TRACK-IK solver thereafter. The solver computes a set of position waypoints in order to achieve the desired [X, Y, Z, x, y, z, w] coordinates and consequently these waypoints are passed to Joint Trajectory Controller (Fig. 13). [3]

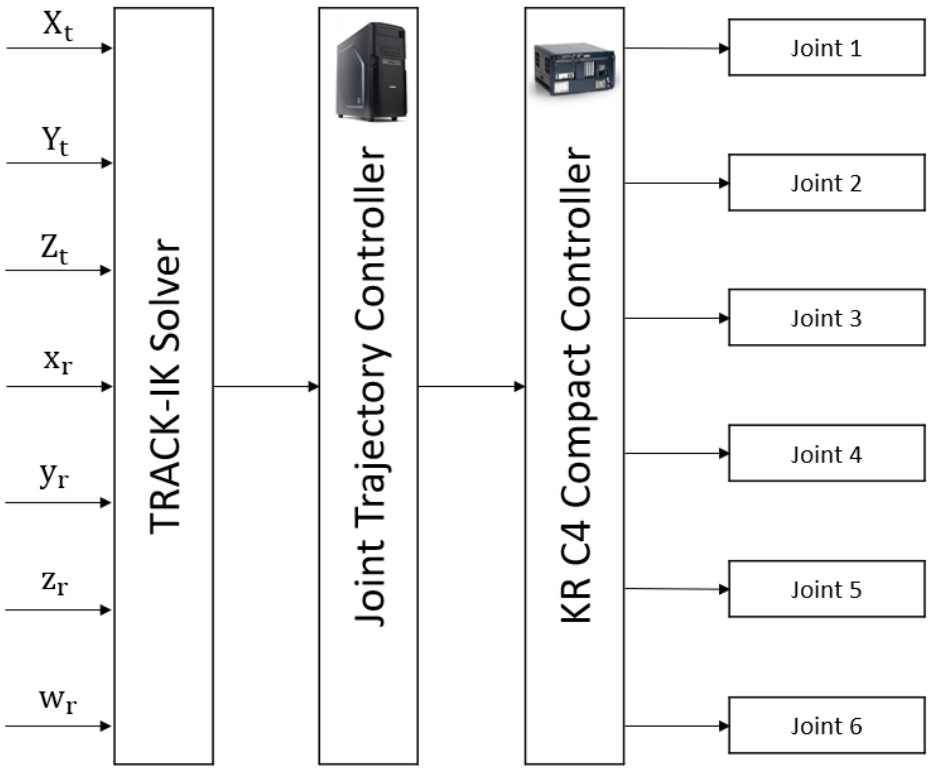

Fig. 13 Motion control architecture

The standard simplest approach consisting of using basic MoveIt API does provide functions that support setting joint goals, setting end-effector goals, creating motion plans etc. However, these functions have two major disadvantages:

1) The robot has to finish one trajectory before starting the new one, which results in the undesirable movement. As the robot is finishing the first trajectory, the movement of the robot is decelerating. At the beginning of the second trajectory the robot is accelerating from the zero speed and in the end of the trajectory it is decelerating again.
2) The second approach is to plan a trajectory before its execution. However, within this thesis the policy does not function as a predictor of trajectory but as the real-time controller that is capable of adapting to new situations.

The strategy from the Fig 13. has been employed because of the reason that it allows to change the trajectory smoothly during its execution without any impact on the robot's movement. The other advantage is that the program does not wait until the robot achieves goal position, thanks to which other computations can be done in the meantime.

5.4. KUKA Reach

For the purpose of testing SAC+HER combination, the "KUKA Reach" task has been created, which consists of controlling KUKA's end-effector to reach the random goal position within the predefined robot workspace. Even though this task does not need to be solved using RL methods, it provides possibilities to learn how to build a custom environment and to verify SAC+HER. Creating this challenge was inspired by OpenAI's FetchReach environment that has been created using MuJoCo physics engine. The initial pose of the robotic arm is constant, whereas the goal position is randomly chosen.

5.4.1. Reward Function

As it was already mentioned in the previous chapters, using dense rewards is not convenient due to difficulties within designing a suitable reward function. As a result, KUKA Reach environment use the following reward function:

$$R(g_{ach}, g_{des}) = \begin{cases} 0 & \text{if } |g_{ach} - g_{des}| < \varepsilon \\ -1 & \text{otherwise} \end{cases}, \tag{45}$$

where g_{ach} denotes the achieved goal, g_{des} is the desired goal and ε represents tolerance. The agent aims to reach a desired goal that is specified by $[X, Y, Z]$ coordinates within the Cartesian coordinate system. After each timestep the reward function compares the achieved goal with the desired goal using subtraction and thereafter compares the absolute value of the difference with the tolerance. Within this task the tolerance is set to 5 mm.

5.4.2. States

A state of a robotic environment is always partially observable and therefore it was necessary to choose properties, which provide sufficient information about the environment, but also their amount would not slow down learning process. After careful consideration the state consists of these ten properties:

1) The pose of the end-effector with respect to the world frame
   - given in 3 positional elements $[X, Y, Z]$
   - given in 4 rotational elements $[x, y, z, w]$ (a quaternion)
2) The position of the gripper with respect to the goal position
   - given in 3 positional elements $[X, Y, Z]$

Another advantage of using ROS and MoveIt is that it is easy to get the aforementioned information without slowing down computational power. The pose of the end-effector is provided by MoveIt, whereas the position of the gripper with respect to the goal position is provided by subtracting the position of the gripper from the goal position.

5.4.3. Actions

The policy outputs three continuous actions representing $[X, Y, Z]$ position coordinates. Each action value is from bounded interval $\langle -1{,}0; 1{,}0 \rangle$. After choosing actions, these action values are converted thereafter into the valid values of the robot workspace. It was necessarily to define the cuboid robot workspace, in which the robotic arm would be capable to reach any point within the workspace. The range of the coordinates has been chosen as follows:

$$\begin{aligned} X_{axis} &\in \langle 0{,}47\ m; 0{,}765\ m \rangle \\ Y_{axis} &\in \langle -0{,}35\ m; 0{,}35\ m \rangle \\ Z_{axis} &\in \langle 0{,}1\ m;\ 0{,}6\ m \rangle \end{aligned} \tag{46}$$

The outputs $[X_\pi, Y_\pi, Z_\pi]$ from the policy π are converted into the physical coordinates $[X_{gripper}, Y_{gripper}, Z_{gripper}]$ using following equations:

$$\begin{aligned} X_{gripper} &= (0{,}1475.X_\pi + 0{,}1475) + 0{,}47 \\ Y_{gripper} &= (0{,}35.Y_\pi + 0{,}35) - 0{,}35 \\ Z_{gripper} &= (0{,}25.Z_\pi + 0{,}25) + 0{,}1 \end{aligned} \tag{47}$$

5.4.4. ROS Mechanism

The best practice within designing a RL system is to divide the whole system into two parts: agent and environment. Fortunately, the architecture of ROS is based on distributed computing, where the whole system consists of small programs that communicate with one another. Despite the faster speed of the whole system, even though one of the programs dies, the other programs continue running. In terms of creating the RL system this advantage comes in handy.

The agent and environment are two separate ROS nodes that communicate together through ROS Services. Services are two-way communications that are based on request/reply interaction. One node, the client, sends a request to the second node, the server. The program in the first node waits for the response from the server. The agent fulfils the role of the client, whereas the environment functions as the server. No matter how long the server/environment is working on the response, the client/agent is blocked until the response comes. In other words, the environment indicates the size of one timestep.

The standard approach for creating a RL environment consists of two main functions. The first function should somehow reset the environment that is necessarily to call before each episode. Its purpose is to prepare the environment for new episode: define new goal, reset all variables, set a robot into an initial pose, etc. The second "step" function should take an action and return a new state of the environment, rewards and other useful information. Regarding the thesis, the content of the request message, which the client sends to the server, defines which action should be called. The custom request message has the following structure:

1) action – the vector of three elements representing $[X_\pi, Y_\pi, Z_\pi]$
2) reset – the flag that differs between the reset function and the step function

and the response message looks as follows:

1) state – the vector of ten elements representing a state of the environment
2) desired goal – the vector of three elements given in the positional coordinates
3) achieved goal – the vector of three elements given in the positional coordinates
4) reward – the scalar value
5) done – the flag representing the terminal state

At the beginning the agent sends a request with the reset flag set to 1. As a result, the environment ignores the received actions and prepares itself for new episode. After setting the robot into the initial pose, it creates the new desired goal and consequently it responds with the current state, … etc (the response message) to the agent. Based on the response message, especially based on the state and the desired goal, the agent takes an action and the episode has officially started. By taking the action, the new request is sent to the environment but with the reset flag set to 0. The environment knows that it should be the step function and therefore it converts action to the physical coordinates (Eq. 47). The fixed orientation of the gripper and the desired $[X_{gripper}, Y_{gripper}, Z_{gripper}]$ coordinates are assigned to the TRACK-IK solver in order to compute a set of position waypoints (Fig. 13) that are applied to the robot for the time of 30 ms. Since the action needs certain time to have an impact, the new state of the environment is recorded after those 30 ms and consequently sent back with other information to the agent.

The new desired goal is randomly generated in the robot workspace after calling the reset function. After that its coordinates are broadcasted to the tf2 library with the intention of knowing transforms between the goal and other frames. For the purpose of visualising the goal's coordinates, they are also published to RViz and Gazebo. In terms of Gazebo the goal is represented by an object without collision setting so it could not interact with other objects, such as the robot. However, in order to be able to set the position of the object, it is necessarily to create a custom

Gazebo plugin for the object that subscribes the goal's coordinates topic. In terms of RViz the goal is visualized using a simple marker. In addition, for better overview the custom RViz plugin has been created so it would be easier to follow the learning process.

The simulation within Gazebo behaves almost as the real hardware in terms of using the same high-level controllers and having almost precise physical properties. Moreover, Gazebo allows to speed up the simulation time by multiplying the real time factor. Luckily, Gazebo is strongly tied with ROS which results in inheriting the simulation time by other ROS nodes. In practice it means that even though there is waiting period of 30 ms, it is scaled by the real time factor. With respect to the thesis and the Reach task, this feature accelerated the learning process 8 times. The maximum real time factor varies hugely, depending on the computing power.

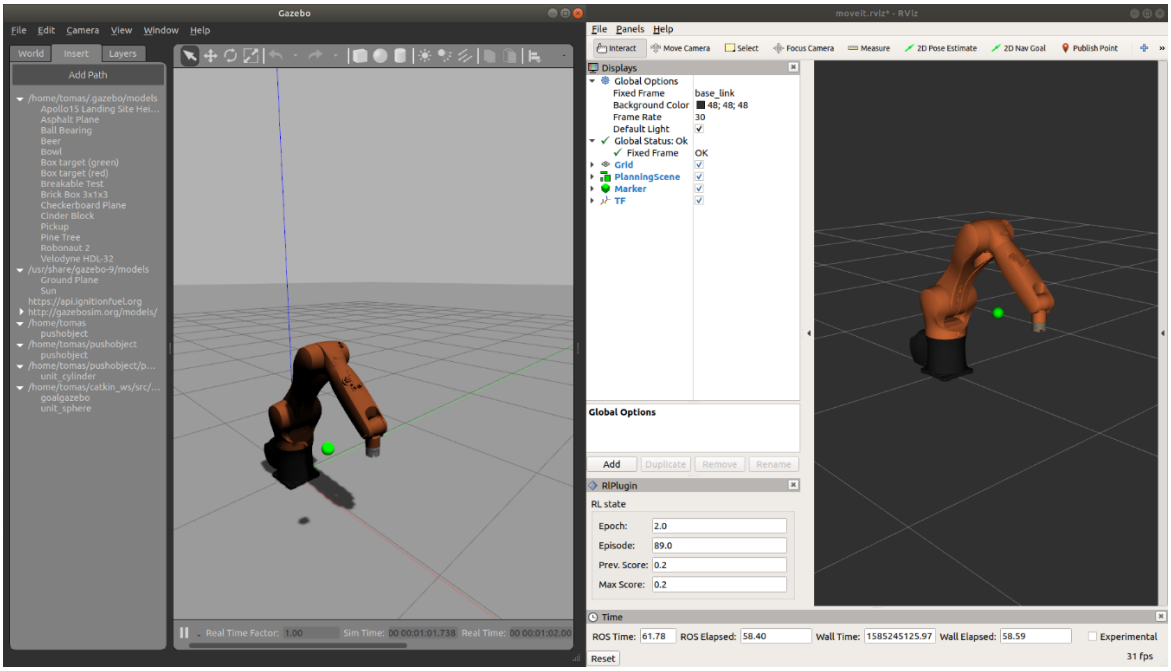

Fig. 14 KUKA Reach environment

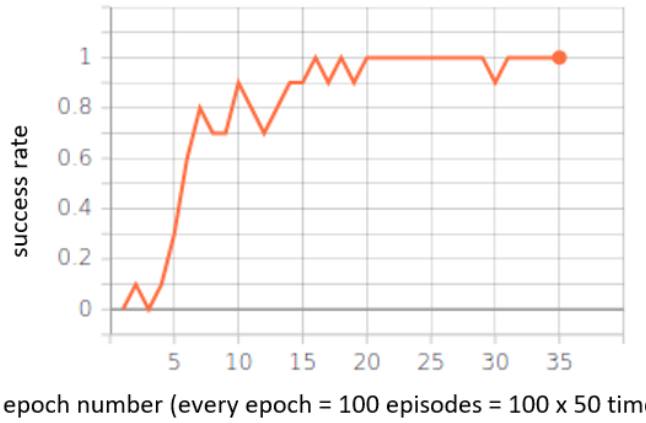

Fig. 15 Detailed learning performance of SAC+HER in KUKA Reach environment

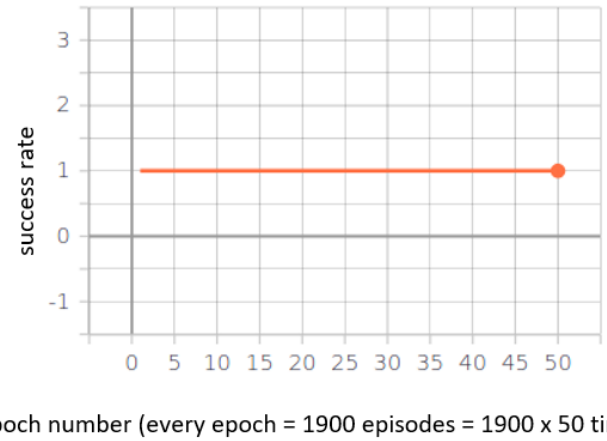

Fig. 16 Long-term learning performance of SAC+HER in KUKA Reach environment

5.5. KUKA Push

The second real world robotic task involves training the KUKA robot to push a block towards the goal position, which is randomly chosen within the predefined robot workspace. Extending the original KUKA Reach task with the block brings more challenges within RL but also within ROS capabilities. In order to detect the pose of the object, the Kinect One sensor has been added to the robot system. The Kinect sensor sends RGB images to the PC via USB 3.0 connection with the frame rate up to 30 FPS. Based on these RGB images and ArUco markers technology the PC is capable to locate the object. In this section the requirements for creating the KUKA Push environment and testing the agent are described.

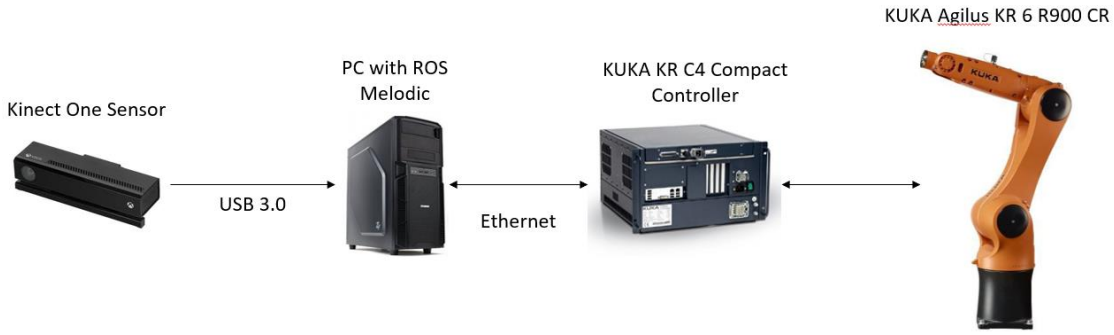

Fig. 17 System overview extended with Kinect One sensor

5.5.1. Kinect v2 Sensor

The Kinect One sensor is a motion sensing input device produced by Microsoft. Even though it did not appeal a gamming community, it has ~~been~~ found applications within amateur robotics. Kinect is equipped with a colour camera and a depth sensor including IR camera and IR projector, and therefore it is referred to as a RGB-D camera. Combination of the inputs from those sensors can produce pointclouds that create a representation of a 3D object with densely placed points along its surface.

Color	**Camera Resolution** **Framerate**	1920 × 1080 pixels 30 frames per second
Depth	**Camera Resolution** **Framerate**	512 × 424 pixels 30 frames per second
Field of VIEW (depth)	**Horizontal** **Vertical**	70 degrees 60 degrees
Operative Measuring Range		from 0.5 m to 4.5 m
Depth Technology		Time-of-flight (ToF)
Tilt Motor		No
USB Standard		3.0

Fig. 18 Technical specifications of Kinect v2

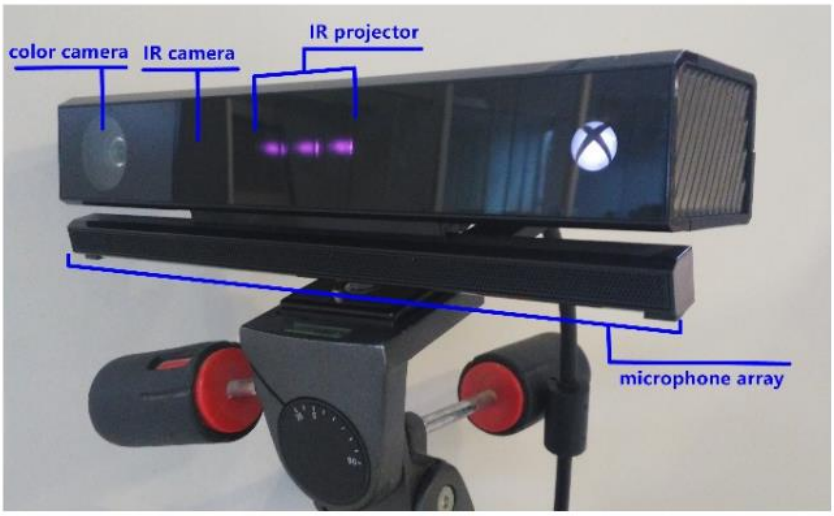

Fig. 19 Kinect v2 description

5.5.1.1. Camera Calibration

The fact that Kinect is not used for robotic applications and is made from cheap pinhole cameras results in significant distortion. This distortion causes that images do not correspond to the real world in terms of the relation between pixels and the real-world units e.g. millimetres. Additionally, even though objects are not moving, their detected coordinates are not stable. Luckily, a calibration can solve the distortion by finding appropriate coefficients within correction equations. The calibration is executed using various types of patterns, mostly a black-white chessboard pattern that was also used in this thesis. Basically, it is necessarily to take photos of the pattern with different orientations of the pattern and different distances between Kinect and the pattern. After taking many photos, the calibration program determines the essential correction coefficients.

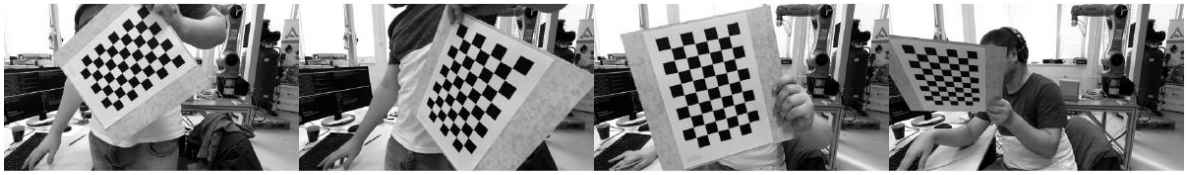

Fig. 20 Camera calibration

5.5.1.2. Hand-Eye Calibration

Despite having the successfully calibrated camera the program would still detect coordinates of an object with respect to the camera. Therefore, there are two crucial parts to do. Firstly, the precise position of the camera must be known relative to the robot's base. Even small error within this step will cause inaccuracy and the robot will not be able to interact with objects because of the wrong position estimates. The second important part is to transform the coordinates, which are originally relative to the camera, with respect to the robot's base. Fortunately, the tf2 library can take care of the second part.

The solution for the determining the camera coordinates is based on the publication by R. Tsai from 1989 and ArUco markers. Tsai's hand-eye calibration can be done for two configurations. The first configuration is called "eye-on-hand", in which case the camera is mounted on the robot's end-effector and the reference frame (marker) is on a table. However, this configuration is not suitable for the KUKA Push application because in case that the object would be far from the end-effector, the camera would not be able to see the object. The second configuration is called "eye-on-base", in which case the camera is standing on a tripod next to the robot and the reference frame (marker) is mounted on the robot's end-effector. With the predefined size and type of an ArUco marker the detection program can determine its position and orientation even only from RGB images. The pose of the end-effector is known thanks to MoveIt and so the real pose of the reference frame is known too. Thereafter, using the "easy_handeye" package the arm is moving into random poses, each time a different rotation and position, and in each pose the actual pose of the reference marker is compared with the output from the detection program. This package takes care of determining the pose of the camera with respect to the base of the robot as well as of all necessary steps needed for the calibration.

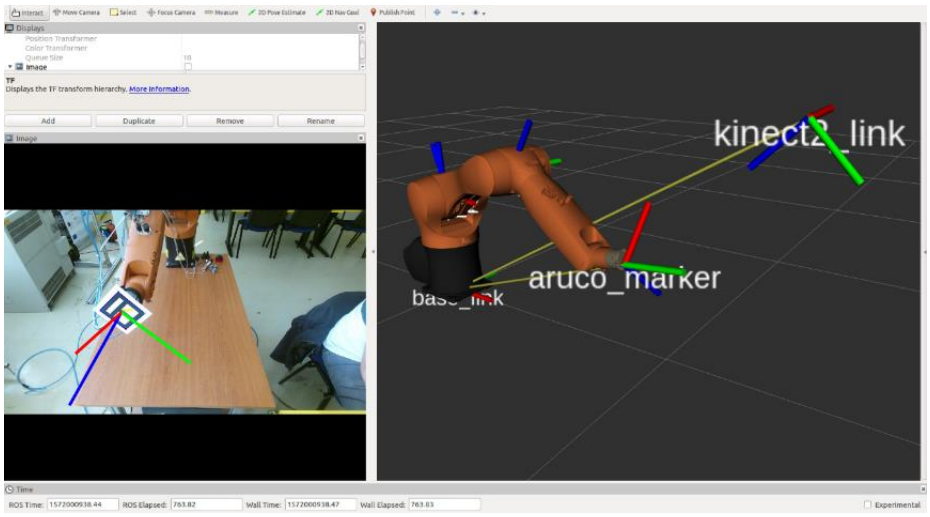

Fig. 21 Eye-on-base calibration of Kinect

5.5.1. Reward Function

The reward function for the KUKA Push task is almost the same as it is in the previous task. The difference is that the desired goal is a goal position g_{des} of a block and the achieved goal g_{ach} is an actual position of the block. The tolerance ϵ is set to 1 cm (Equation 45). In this task the aim for the robot is to push a block towards a goal position using its end-effector. The goal position of the block and the initial block position are randomly chosen on the table surface within the robot workspace.

5.5.2. States and Actions

The state space within this task is 26-dimensional. The properties of the state represent the following features:

1) The pose of the end-effector relative to the world frame
   - given in 3 positional elements $[X, Y, Z]$
   - given in 4 rotational elements $[x, y, z, w]$
2) The pose of the block relative to the world frame
   - given in 3 positional elements $[X, Y, Z]$
   - given in 4 rotational elements $[x, y, z, w]$
3) The position of the block relative to the end-effector
   - given in 3 positional elements $[X, Y, Z]$
4) The linear velocity of the block
   - given in 3 linear velocity elements $[v_x, v_y, v_z]$
5) The angular velocity of the block
   - given in 3 angular velocity elements $[w_x, w_y, w_z]$
6) The linear velocity of the end-effector
   - given in 3 linear velocity elements $[v_x, v_y, v_z]$

The pose of the end-effector is obtained using MoveIt and its linear velocities are computed by means of the actual and previous position of the end-effector. The access to features of the block depends on the fact if the Gazebo simulator or the real physical robot is used. In case of using Gazebo, the pose and velocities are defined by the custom plugin that was created for the purposes of publishing the properties of the object. The publishing rate is 1000 Hz, whereas the sampling time needed for computing velocities is 40 ms (the sampling rate is 25 Hz). This sampling time is defined by the size of the timestep within RL interactions. In case of using the real physical robot, the object pose is determined by the Kinect sensor and an ArUco marker. Velocities are computed in the same way like the linear velocities of the end-effector, which means by using the actual and

previous poses of the object. The sampling time and the RL timestep are defined by time needed for Kinect and the object recognition program to detect an object pose.

Actions from the policy specify target $[X, Y, Z]$ coordinates of the end-effector, which are published to Joint Trajectory Controller. All actions are in the interval $\langle -1.0; 1.0 \rangle$ and after choosing an action they are consequently converted into the valid values within the robot workspace. Coordinates within the robot workspace are bounded as follows:

$$\begin{aligned} X_{axis} &\in \langle 0{,}6\ m; 0{,}98\ m \rangle \\ Y_{axis} &\in \langle -0{,}34\ m; 0{,}34\ m \rangle \\ Z_{axis} &\in \langle 0{,}43\ m;\ 0{,}63\ m \rangle \end{aligned} \tag{48}$$

5.5.3. ROS Mechanism

The core of the program for the KUKA Push task is essentially the same as the one for KUKA Reach. In other words, the KUKA Push task can be considered as the extension of KUKA Reach. The main difference is that the block (a cube 4x4 cm) is added (Fig. 21). The training phase is done in Gazebo and therefore it was necessary to develop the custom plugin for it. The plugin functions as a publisher and a service server as well. The publisher is responsible for reading and sending object pose and velocities each 1 ms, whereas the service server responds to the environment request for resetting the object position. The object position is randomly chosen after calling the reset function within the environment. The rest of the system remains the same as it is in case of KUKA Reach, except the fact that the state vector in the agent's request has 26 elements. The following figure shows the overview of the ROS mechanism.

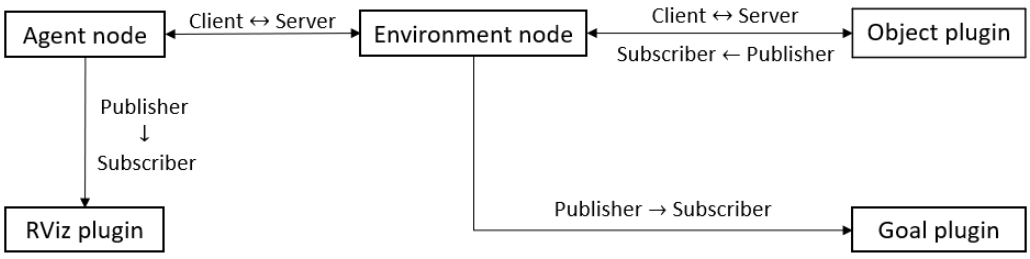

Fig. 22 Overview of ROS mechanism for KUKA Push

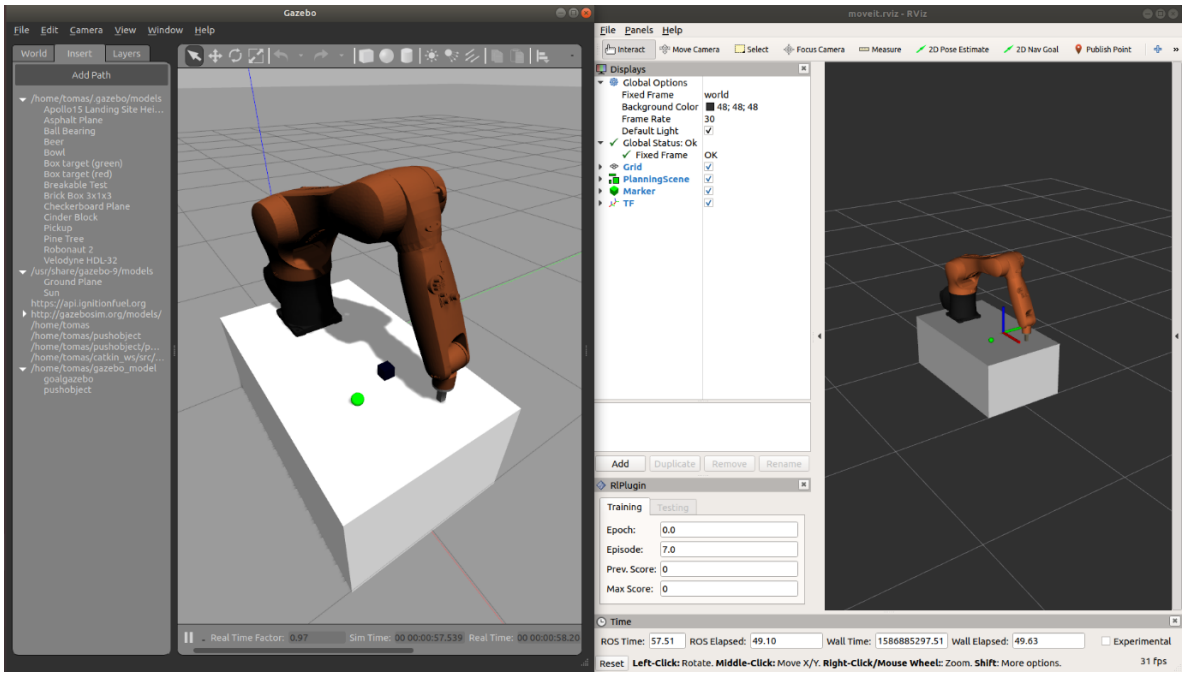

Fig. 23 KUKA Push environment

Fig. 24 Learning performance of SAC+HER in KUKA Push environment

Conclusion

The aim of this thesis was to verify the Soft Actor-Critic and Hindsight Experience Replay algorithms in the custom created environments within ROS. The prerequisite for accomplishing this challenge was to understand the basics of reinforcement learning, moreover, the SAC and HER algorithms in order to be capable of implementing them within ROS. The combination was tested using OpenAI Gym environments [25] and modified until the learning performance was satisfying. These tests showed that using automating entropy adjustment significantly improves the learning performance in terms of speed and convergence stability.

The next phase was to create the robotic environment KUKA Reach. Since this task is relatively simple to solve, its development provided more space for considering different approaches how to solve inverse kinematics, motion planning, etc. For the purposes of this thesis using Joint Trajectory Controller from *ros_control* package and the TRACK-IK solver was the best way to go. Basically, the agent determines $[X, Y, Z]$ coordinates of the end-effector and MoveIt with *ros_control* takes care of the waypoints between the current position and agent's request. The best advantage of using this approach is that the agent is able to change the trajectory smoothly without any negative impact on the robot movement. The KUKA Reach environment provides solid foundation for creating other robotic environments within ROS. However, the KUKA Reach task does not need RL in order to be solved and therefore the KUKA Push environment has been developed. Because the environment is more complex task, it brought more complications. The first main problem was in using the original tf library for detecting the pose of the object. The tf library had sometimes problem to get the current state of the object, which resulted in killing the learning program. This problem was solved by using the tf2 library that is a newer version, although in the final version of program the tf2 library was replaced by publisher/subscriber. The second issue concerned the shape of the gripper. The cylindrical gripper caused that the object was thrown away from the robot workspace each time the gripper hit some edge of the object. Changing the original cylindrical gripper with the cuboid gripper representing a finger has resolved this issue. Choosing the right features, which input to the policy, was also problematic and it is necessary to realize what the agent really needs to know in order to solve a task. Regarding the thesis, velocities of the object and the gripper were crucial conditions for accomplishing the KUKA Push task. The final main problem was to define the robot workspace and initial positions of the object and a goal position. The agent could not learn if the space of initial positions was too large.

The simulation results of the KUKA Push task prove that the combination of SAC+HER has state-of-the-art performance within robotic environments. Unfortunately, there was not enough time for

testing the trained agents on the real aforementioned hardware so the next logical step should be verifying the agents in the real world. Besides that, the future work could have two options. The first option would be to use this combination for solving more difficult robotic environments using ROS, such as handling of compliant food objects, or trying to achieve even better performance by using LSTM neural networks. The other option would be to work on new types of algorithms and test it on the environments that have been created within the thesis. The algorithm that seems to be particular interesting is the Monte Carlo Tree Search algorithm for continuous action and state space. This approach is still new and requires a lot of work, but it would be interesting to see how well it performs within robotics. Anyway, the created open-source environments could hopefully enable ROS community to test different RL algorithms or motivate its members to create other environments.

Bibliography

[1]. Achiam, Josh and Abbeel, Pieter. OpenAI Spinning Up. [Online] OpenAI, 2018. [Cited 1. December 2019.] https://spinningup.openai.com/en/latest/index.html.

[2]. DARPA Robotics Challenge. [Online] DARPA. [Cited: 16. 4. 2020.] https://www.darpa.mil/program/darpa-robotics-challenge.

[3]. Ďurovský, František. DARPA - Robotics Challenge 2015. Smart Robotic Systems. [Online] 12. September 2015. [Cited: 16. April 2020.] http://www.smartroboticsys.eu/?p=1986.

[4]. Levine, Sergey. Deep Robotic Learning. [Online] 7. April 2017. [Cited: 16. April 2020.] https://youtu.be/eKaYnXQUb2g.

[5]. Kober, Jens, Bagnell, J. Andrew and Peters, Jan. Reinforcement learning in robotics: A survey. 11, s.l. : The International Journal of Robotics Research, 2013, Vol. 32.

[6]. Grondman, Ivo. Online Model Learning Algorithms for Actor-Critic Control. 2015. ISBN 978-94-6186-432-1.

[7]. Sutton, Richard S., Barto, Andrew G. Reinforcement Learning: An Introduction. s.l. : The MIT Press, 1998. ISBN 0-262-19398-1.

[8]. Silver, David. Introduction to reinforcement learning. [Online] DeepMind, 13. May 2015. [Cited: 16. April 2020.] https://youtu.be/2pWv7GOvuf0?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ.

[9]. Henderson, Peter et al. Deep Reinforcement Learning that Matters. New Orleans : The Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[10]. Haarnoja, Tuomas et al.. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv preprint, 2018. arXiv:1801.01290.

[11]. Simonini, Thomas. Deep Reinforcement Learning Course. [Online] 2018. [Cited: 16. April 2020.] https://simoninithomas.github.io/Deep_reinforcement_learning_Course/.

[12]. Haarnoja, Tuomas et al. Reinforcement Learning with Deep Energy-Based Policies. s.l. : arXiv preprint, 2017. arXiv:1702.08165.

[13]. Géron, Aurélien. A Short Introduction to Entropy, Cross-Entropy and KL-Divergence. [Online] 2. February 2018. [Cited: 16. April 2020.] https://youtu.be/ErfnhcEV1O8.

[14]. Beeson, Patrick a Barrett, Ames. TRAC-IK: An Open-Source Library for Improved Solving of Generic Inverse Kinematics. Seoul : IEEE RAS Humanoids Conference, 2015.

[15]. Jiao, Jichao et al. A Post-Rectification Approach of Depth Images of Kinect v2 for 3D Reconstruction of Indoor Scenes. 11, s.l. : International Journal of Geo-Information, 2017, Vol. 6.

[16]. Schulman, John et al. High-Dimensional Continuous Control Using Generalized Advantage Estimation. s.l. : arXiv preprint, 2015, arXiv:1506.02438.

[17]. Andrychowicz, Marcin et al. Hindsight Experience Replay. s.l. : arXiv preprint, 2017, arXiv:1707.01495

[18]. Plappert, Matthias et al. Ingredients for Robotics Research. OpenAI. [Online] 26. February 2018. [Cited: 16. April 2020.] https://openai.com/blog/ingredients-for-robotics-research/

[19]. Fujimoto, Scott et al. Addressing Function Approximation Error in Actor-Critic Methods. s.l. : arXiv preprint, 2018, arXiv:1802.09477

[20]. Ďurovský, František. Vision system for robot-human cooperation. 2017. Technical University of Košice.

[21]. Duan, Yan. Sampling-based Approximations and Function Fitting. [Online] 5. October 2017. [Cited: 16. April 2020.] https://youtu.be/qO-HUo0LsO4

[22]. Mnih, Volodymyr. Deep Q-Networks. [Online] 5. October 2017. [Cited: 16. April 2020.] https://youtu.be/fevMOp5TDQs

[23]. Haarnoja, Tuomas et al. Soft Actor-Critic Algorithms and Applications. s.l. : arXiv preprint, 2018, arXiv:1812.05905

[24]. KR AGILUS. [Online] KUKA. 6. September 2017. [Cited: 16. April 2020.] https://www.kuka.com/sk-sk/produkty-a-slu%C5%BEby/robotick%C3%A9-syst%C3%A9my/industrial-robots/kr-agilus

[25]. Gym RL environments. [Online] OpenAI. [Cited: 16. April 2020.] https://gym.openai.com/

Appendices

Appendix A: DVD with electronic version of the thesis, videos and source codes

Appendix B: Hyperparameters

Appendix B: Hyperparameters

Parameter	Value
optimizer	Adam
learning rate	10^{-3}
discount (γ)	0,99
replay buffer size	10^6
number of hidden layers (all networks)	2
number of hidden units per layer	256
number of samples per minibatch	256
nonlinearity	ReLu
polyak hyperparameter (ρ)	0,995
target entropy	-4