DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reinforcement Learning for Path Planning of Robotic Arms

Anton Mai

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reinforcement Learning for Path Planning of Robotic Arms

Bestärkendes Lernen zur Pfadplanung von robotischen Armen

Author:	Anton Mai
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Dr. rer. nat. Zhenshan Bing
Date:	February 17, 2020

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 2. April 2020 Anton Mai

Abstract

Robotic Arms are used in many fields due to their high accuracy. A robotic arm has the advantage of being able to solve the same tasks like a human arm because of its similar structure, and being able to work repetitively and independently of any human. Setting up the task of a robotic arm requires the engineer to plan a path for solving the task. Using an approach with reinforcement learning to make the robotic arm learn the path independently has not been feasible due to the sparse rewards of robotic arm tasks. A recently successful method of using Reinforcement Learning is the concept of Hindsight Experience Replay.

This thesis extends the difficulty of existing tasks to two new tasks and examines the performance of Hindsight Experience Replay on these tasks. In the first experiment the robotic arm has to move a ball towards a point that is far outside of its reach, similar to golf. In the second experiment the task is to toss a ball into a box that is also outside of its reach, similar to basketball.

Results show that vanilla Hindsight Experience Replay performs poorly on these tasks. Further research on solving the tasks with improvements to Hindsight Experience Replay, like Hindsight Goal Generation and Energy-Based Hindsight Experience Prioritization is required to make further progress on solving these tasks.

Contents

Abstract **vii**

List of Abbreviations **xi**

1. Introduction **1**

2. Theoretical Background **5**
- 2.1. Reinforcement Learning 5
- 2.2. Artificial Neural Networks 8
- 2.3. Deep Deterministic Policy Gradients 9
- 2.4. Hindsight Experience Replay 11

3. Methodology **13**
- 3.1. Simulation Environment 13
  - 3.1.1. MuJoCo 13
  - 3.1.2. OpenAI 14
- 3.2. Model 15

4. Experiments **17**
- 4.1. Fetch environments by OpenAI 17
  - 4.1.1. FetchReach 18
  - 4.1.2. FetchPush 18
  - 4.1.3. FetchSlide 19
  - 4.1.4. FetchPickAndPlace 20
  - 4.1.5. Discussion of Results 21
- 4.2. FetchSlideball 23
  - 4.2.1. Task Description 23
  - 4.2.2. Environment 23
  - 4.2.3. Results 23
  - 4.2.4. Discussion 27
- 4.3. FetchToss 28
  - 4.3.1. Task Description 28
  - 4.3.2. Environment 29
  - 4.3.3. Results 29

4.3.4. Discussion 30

5. Conclusion and Future Work **33**

A. Simulation Parameters **35**

Bibliography **39**

List of Abbreviations

RL	Reinforcement Learning
HER	Hindsight Experience Replay
ANN	Artificial Neural Network
DOF	Degrees of Freedom
DDPG	Deep Deterministic Policy Gradients

1. Introduction

Lee Sedol, one of the best Go players in the world, was beaten by the Go engine AlphaGo in a match with a score of 4 to 1. The engine was clearly stronger [2]. AlphaGo learned from a big dataset of games and got stronger only by playing with itself. Artificial Intelligence is quite popular nowadays because of its many use cases: self-driving cars, playing Atari games, robotics and more. But how do these engines learn how to get so good at their areas? The answer lies in reinforcement learning (RL), an area of machine learning.

The idea of RL is to have a state that an agent is in and actions that it can choose from. Each action results in different states and amounts of earned values that are called rewards. Rewards are used by agent to measure how good an action was. This process is repeated which results in the agent learning which actions in each state are better. Imagine you are a soccer player. You are standing in front of the goal (which is the state you are in). You can either shoot or pass the ball (which are your available actions). You choose to shoot, but the ball is blocked by the goalkeeper (you get a low reward). So the next time you are in front of the goal again, you are more likely to try to pass the ball to a teammate. This time your teammate scored a goal (you got a high reward). From these experiences you learn that it is probably better to pass the ball if you are standing in front of the goal. The concept of RL can be used in a variety of environments, for example robotic arms.

Already in the 14th century, Leonardo da Vinci made blueprints of robotic arms [15]. A robotic arm resembles a human arm. It consists of segments which are connected by joints [10]. The number of joints correspond to what is called Degrees of Freedom (DOF). A robotic arm with 5 joints would have 5 DOF because it can pivot in 5 ways. Each joint is connected to a step motor. Step motors supply energy needed to the robotic arm and make the robot move very precisely. The equivalent to a human hand is the end effector. The end effector can vary depending on the task that the robotic arm has to solve.

Robotic arms have many advantages. They are very accurate and consistent which is why they are mostly used for repetitive tasks or tasks that require high accuracy which are hard for humans [1]. This is the main reason why they are used in laboratories and hospitals for surgeries. Being able to work automatically without any human makes robotic arms useful in manufacturing and assembly lines.

Humans still have to teach the robotic arms how to move when setting them up. For path planning of a robotic arm, a sequence of actions has to be found that solves the task. This sequence is saved and repetitively executed by the robotic arm. Finding the path still requires human labor. To improve path planning, many complicated algorithms have been developed. For example, Klanke et. al [11] developed a dynamic path planning algorithm for a 7 DOF Robotic Arm. They solved it by reducing the task to a 6 DOF problem, which is easier to solve. A robotic arm needs 6 DOF to be able to move its end effector in every direction and orientation. This also means that robotic arms with more DOF do not have a unique path to solve the tasks. There are different paths which can vary in length and energy consumption. To improve the quality of the path and to do path planning without a human, using RL for robotic arms is a logical approach.

There is an issue that prevents robotic arms to learn with RL. It is hard to construct a suitable reward function for tasks where robotic arms are used. So either a suitable reward function has to be constructed which is time and cost consuming, or the simplest reward function, a binary and sparse reward function has to be used. Both approaches have some issues.
Constructing a reward function can be quite complicated. Also, for each task an individual reward function has to be made. So someone has to do this work which defeats the purpose of using RL for robotic arms over path planning by hand. Depending on the case it might be easier to just plan the path without RL.
Using only a sparse reward for robotic arms works as follows. A reward is given if the goal is reached, and no reward is given if the goal is not reached. Robotic arms usually have many DOF, so there is a huge action space for the robotic arm. It is quite unlikely for robotic arms to fulfill the task by doing random movements. Tasks like moving an object are near impossible to solve with random actions. So it is very unlikely for the robotic arm to earn a reward and learn. It takes a very long time to train a robotic arm with sparse rewards. But recently hindsight experience replay (HER) has been introduced by Andrychowicz et al. [3]. HER allows a high learning rate even with sparse rewards by making the training samples more efficient.

HER is inspired by the human ability of not only learning from successes, but also learning from failures [3]. After each episode of training, the actions taken and the states that the agent was in are saved to a Replay Buffer. In case of a failure, the terminating state and the goal state are different, so the earned reward is negative. When replaying the episodes in the Replay Buffer, the goal will be replaced by the terminating state or a state that is close to it. Therefore, when replaying that episode, the agent will be successful and learn how to reach that state. By doing this, the agent will not learn how to reach the goal that was desired, but it will learn another goal which might be helpful in learning how to reach the desired goal. Andrychowicz et al. [3] researched that HER is especially usable for tasks with multiple possible goals, but also for tasks with a single goal they showed that the learning performance improved.

There are many endeavors to improve HER. Most of the improvements are based on the selection of experiences that should be replayed. This is based on the idea that some experiences are more valuable to learning the task than other.
Zhao and Tresp propose a curiosity-driven experience prioritization approach in which rarer experiences are rated more valuable and are therefore prioritized for hindsight replay. [27]
Fang et al. propose a similar approach with their "Curriculum-guided HER". At earlier stages of training, curiosity-driven experience prioritization is used to enforce exploration of the potential approaches to solve the task. At latter stages experiences with higher proximity to the actual goals are prioritized to reinforce actually finding paths to solve the task. [9]
Zhao and Tresp also propose an energy-based Prioritization of Hindsight Experiences. They hypothesized that episodes with higher trajectory energy are more useful for learning than those with lower energy. Their experiments have shown that this approach improves performance and sample-efficiency over state-of-the-art approaches. [26]
Ren et al. introduce Hindsight Goal Generation, an approach that modifies the replay buffer to generate goals that are easy to achieve but still valuable to guide the agent to learn the actual goal. [19]
Lank and Wu propose ARCHER, a modification to HER. They explain that replaying an experience is biased, because the modified new goal would influence the actions of the agent and by forcing the agent to take the same actions, these actions are biased. So there would be a difference between replaying a hindsight experience and a real experience, because in the real experience, the agent might choose a different action because it is not forced. To counter this bias, they use aggressive/high rewards for hindsight experiences so that the agent is more likely to do the same action in the same situation in a real experience. [13] Most of the research on HER revolves around improving the performance of HER using simple environments.

In this thesis the performance of HER is examined for harder environments. Two environments which are prototypes of golf and basketball, will be used to experiment with. In the first environment, the task is for a robotic arm to roll a ball to a point that is far outside of its reach. In the second environment the robotic arm has to toss a ball into a box.

This thesis is structured as follows: Chapter 2 describes the theoretical background on RL in general, artificial neural networks (ANN) the RL algorithm deep deterministic policy gradients (DDPG) and HER. Chapter 3 explains the methodology used for this thesis. The simulation environment is showcased. In chapter 4, the experiments are presented and the results are discussed. In the last chapter, the results are summarized and suggestions for further work is provided.

2. Theoretical Background

This chapter explains the concepts needed to understand this thesis. The theory behind RL, ANN, DDPG and HER is explained.

2.1. Reinforcement Learning

RL is one of the main learning concepts of Machine Learning next to Supervised Learning and Unsupervised Learning [20].
In Supervised Learning some input data is given to the learning agent. The agent is expected to come up with some output which is then compared with the expected output. If the output given by the agent and the expected output matches, then the agent was correct. The agent is trained by calculating the error between expected output and actual output. An use case for Supervised Learning is sorting mails into regular mail and spam mail. The agent is given a mail and it should decide whether the mail is regular or spam based on the content. Supervised Learning is used for classification and regression problems [20]. It is mainly useful when the expected output is already known, so the learning agent can learn to do recognize these.
In Unsupervised Learning there is no expected output. The learning agent is fed with data so it can figure out interesting features and similarities between different data. Unsupervised Learning is often used to cluster data, often pictures, based on similarities [20].
In RL the agent is learning through rewards that are given through interaction with an environment. The goal of a task is clear, but the path of actions to reach the goal is not trivial. RL is used to find the best action in each situation. It is often used for games because they are already set up to have a clear task and goal, but the optimal way to reach it is not clear. Games usually provide full information on the environment in contrast to the real world, therefore setting up the learning environment is simple. Also Supervised Learning is limited in performance because the agent can only learn to become as good the expected output that we set. So in games like chess, with Supervised Learning the agent can only become as good as the best players it learns from, but not better. It is limited by the skills of the best humans [21]. However, RL is not restricted by these limitations. The agent can improve on its own only by exploring his options. In games like

Go and Chess, engines that use RL have already far surpassed the best human players [22].

This section will explain the theory behind RL. The post by Lilian Weng is recommended as a resource on more precise explanations on RL [25]. RL is usually modeled as a Markov Decision Process. The Markov Decision Process for RL consists of following elements [7]:

- A set of states S
- A set of actions A
- The transition probability $P_a(s, s')$ from state s to s′ under action a
- The immediate reward $R_a(s, s')$ of that transition
- rules that describe the agents observation

The agent and environment are in a state s. The agent chooses an action a from its set of possible actions A to interact with the environment. The environment reacts by transitioning to another state and returning a reward R and an observation to the agent. Depending on the model the transitions might be stochastic or deterministic. The aim of the agent is to earn the maximal total reward possible. In order to reach this aim, the agent interacts with the environment to gain knowledge about the environment through the gained rewards and observations. Through this process, the agent learns in which state which actions are better to gain more reward. This is illustrated by Figure 2.1.

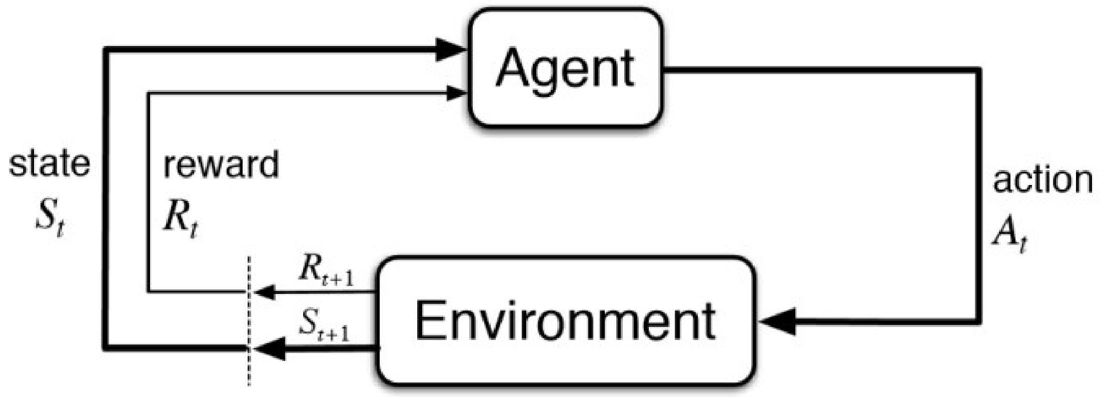


Figure 2.1.: Reinforcement Learning. An agent chooses an action to interact with the environment and gets a reward and an observation of his new state back. [5]

In each state there is an action that the agent considers best due to its current knowledge about the expected rewards of each action. This set of actions is known as the policy

$\pi(s)$. The goal of getting maximal reward can be interpreted as finding the best actions in each state that give the most reward, which is finding the optimal policy. The policy can also be either deterministic or stochastic, depending on the transition probability of the environment.

A value function is used to measure how good a state or action is. Two types of value functions are used for the states and the actions. the state value function is denoted as V(s). The value of a state is the expected reward when acting according to the policy π [7]. V(s) is defined as follows in equation 2.1.

$$V^{\pi}(s) = \mathbb{E}[R|s, \pi] \tag{2.1}$$

The actions value function is denoted as Q(s,a) and is defined In equation 2.2.

$$Q^{\pi}(s, a) = \mathbb{E}[R|s, a, \pi] \tag{2.2}$$

When determining the value of a state or action, a discount factor γ is used to discount future rewards towards immediate rewards. The idea is that a state s is not only as good as the reward you get when transitioning to that state. Future rewards from states that are reachable from state s should also be considered. The value of a state consists of the reward that you get by transitioning to that state and the potential rewards that can be gained by transitioning from that state. Since future rewards are not as certain as immediate rewards, the discount factor is used. The farther a reward is in the future, the more it is discounted. The Bellman equations described in equation 2.3 and 2.4 are a set of equations that convert the value functions into the immediate and future reward and can be used to update the value-functions [25].

$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s)(R(s, a) + \gamma \sum_{s' \in S} P^{a}_{ss'} V_{\pi}(s')) \tag{2.3}$$

$$Q_{\pi}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P^{a}_{ss'} \sum_{a \in A} \pi(a'|s') Q_{\pi}(s', a') \tag{2.4}$$

The goal is to find the actions that return the maximal reward. This is displayed by the Bellman optimality equations (2.5,2.6) [25]:

$$V_{*}(s) = \max_{a \in A}(R(s, a) + \gamma \sum_{s' \in S} P^{a}_{ss'} V_{*}(s')) \tag{2.5}$$

$$Q_*(s,a) = R(s,a) + \gamma \sum_{s' \in S} P^a_{ss'} \max_{a \in A} Q_*(s',a') \tag{2.6}$$

To calculate the optimal values of each state and action, Dynamic Programming could be used if the entire model is known. But even knowing the entire model is not good enough as usually the main issue lies in the huge state and action space, which makes it impossible to use Dynamic Programming. For RL, artificial neural networks (ANNs) can be used to approximate the value functions [16].

When following the current policy, the agent will earn the maximal reward that it could earn with current knowledge, but never more than that. To learn, the agent has to deviate from the policy and explore different actions and states. The question is how much deviation is necessary, as the agent also needs to exploit most of the policy path it has learned until now because following most of the policy has a higher probability of earning a high reward. This is known in RL as the exploration vs. exploitation problem. Usually there is a variable ϵ that determines with which probability the agent will deviate from the policy. It is set rather low to let the agent exploit most of its policy. As an example, an approach is ϵ-greedy [12]. With a high probability of $1-\epsilon$ the agent will choose the policy and with a probability of ϵ a random action is chosen. This ensures that the agent exploits most of the policy, but also explores more possible actions that might bring more reward.

2.2. Artificial Neural Networks

ANNs are inspired by the human brain. The ANN consists of layers of neurons. Each neuron is connected to the next layer of neurons. There is one input layer and one output layer at the beginning and end of the layer of neurons. The layers between the input and output layers are called hidden layer. The hidden layer can consist of only one or more layers. The idea is to train the ANN to take inputs and produce outputs. To approximate the value functions the input would be states and actions, the output should be the correct and optimal values of these states and actions. An example of an ANN is shown in Figure 2.2.

The learning process of the neuronal network is as follows. Each neuron obtains inputs x_i by the the outputs of the neurons in the layer before it. Each input value is weighted and then the sum of these weighted values is taken. A bias b is also added to support the learning process. After using an activation function on the sum, the value is output to the next layer of neurons. The activation function is a simple function that either reduces the output of the neuron to 0 if the value is below a certain threshold, otherwise the value is output unfiltered. The training process of the neuron can be seen in Figure 2.3.

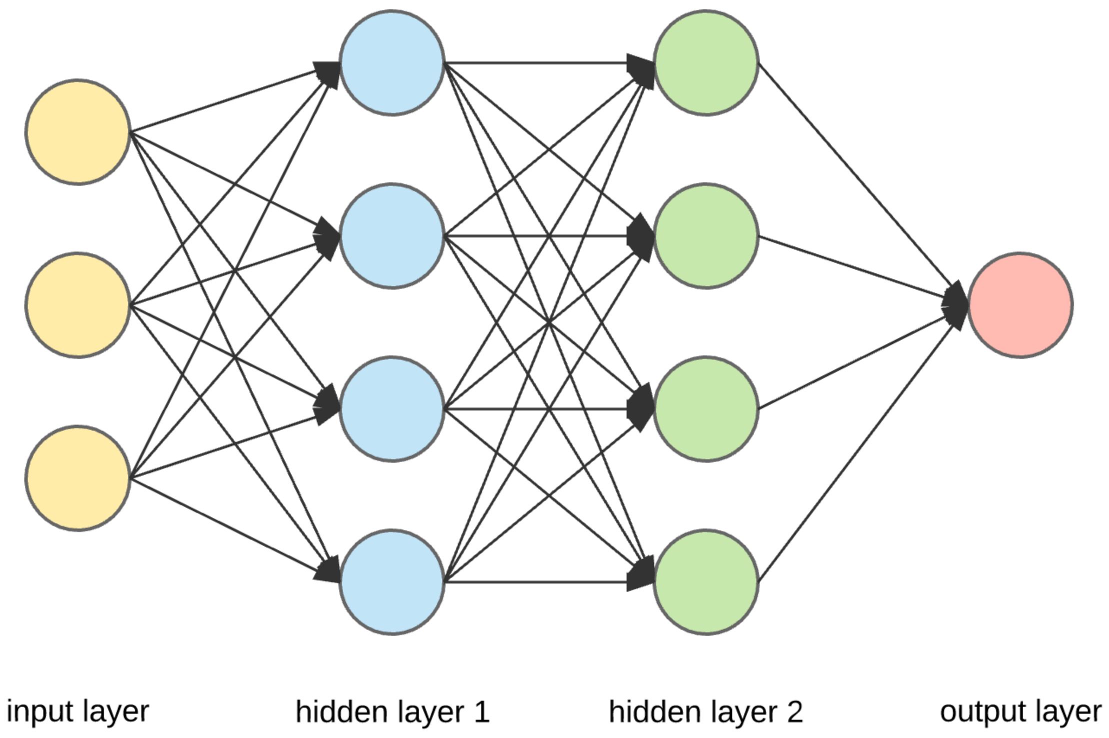


Figure 2.2.: A neural network. [4]

When the ANN outputs a value, a process called Backpropagation is used to improve the ANN [14]. When initializing the ANN, the actual right weights for the input values are unknown, so random values for the weights are used. Therefore, the output values will not be right. By using an error function, the amount of error can be computed. The amount of error can be used to figure out how much the weights have to be modified for the output to become right. Backpropagation is the process of going backwards through the ANN and improving the weights of the neurons that caused the output value to be wrong. By repeating the whole process, the weight of each neuron converges towards an optimal value.

2.3. Deep Deterministic Policy Gradients

The paper by Silver et al. is a recommended resource for more detailed explanations [23]. This chapter will give a short summary of DDPG. The algorithm DDPG learns concurrently a Q-function (the action-value function) and a policy. [23] Q*(s,a) is used to find

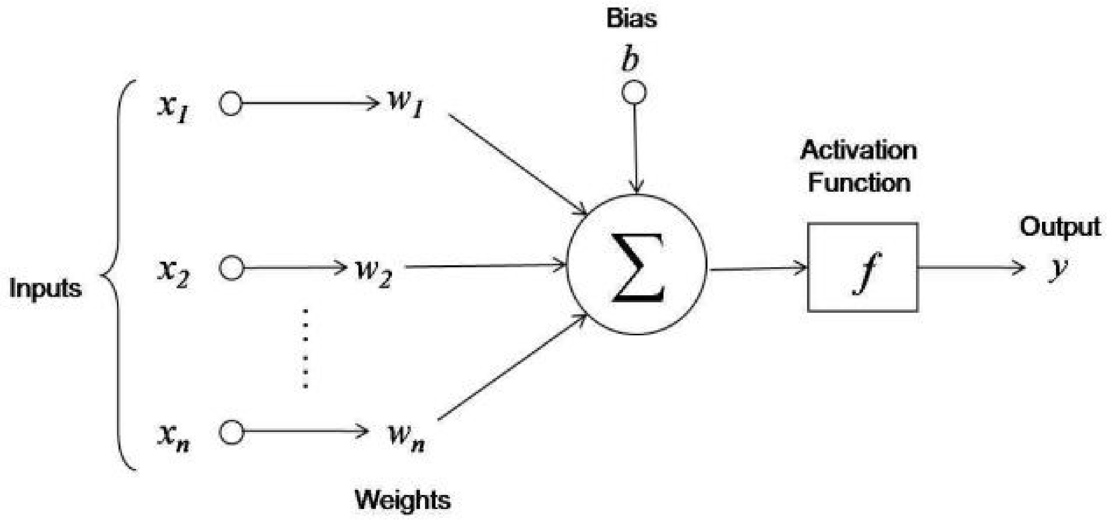


Figure 2.3.: A neuron and how its value is composed [18]

the the optimal action in each state. To compute the Q-values for a discrete action space, the Q-values of each action could be calculated and compared to find the biggest value. But in a continuous action space it is not possible to calculate the Q-value for each action. DDPG uses the fact that the action space is continuous and so Q*(s,a) is expected to be differentiable in respect to the action argument. [23] This way it is possible to approximate the Q-values and policy.

To learn the Q-values, the Bellman equation for the action value is used. To approximate the Q-values, a mean squared Bellman error function is used. The idea is that minimizing this function error is equal to approximating the current Q-values to the optimal Q-values.

For DDPG an experience replay buffer is also used. The replay buffer is a set of experiences. This can be used to replay old experiences. When only using new experiences, the ANN might be overfit to those experiences. Being overfit means that for some experiences the ANN will output very good results, but for most other experiences it will perform very bad. Experience replay is useful to prevent that. But a too large buffer can cause the learning process to slow down. The right balance has to be found.

DDPG also uses target networks. Equation 2.7 is called target [23].

$$r + \gamma(1-d)\max_{a'} Q_{\phi}(s', a') \tag{2.7}$$

The target is what is desired for the Q-function to approximate to. When minimizing the mean squared bellman error function, there is the problem that the target is also dependent on the parameters that are trained. When changing the parameters, the target would also change which is problematic. That is why the target network, a copy of the ANN is used. The update of the target network is delayed to avoid this conflict. To find the optimal policy, simply gradient ascent can be used to find the maximal Q-values.

2.4. Hindsight Experience Replay

Sparse rewards are a big issue in RL, especially in tasks for robotic arms often the rewards are sparse. Having sparse rewards means that most of the samples used for training will not successful and therefore will not bring any useful reward. For example, the task to move an object to a certain point would have a sparse reward for a robotic arm because very precise movements are needed which the robotic arm has to learn first.

Andrychowicz et al. have shown that HER can be used to deal with this issue for robotic arms [3] HER can learn efficiently from sparse rewards and can also be combined with any off-policy RL algorithm. This technique is inspired by the ability of humans to learn from failures as least as much as from successes.

HER works as follows. After an episode of gaining experiences, all transitions between the states in each training sample is stored in a replay buffer, but the goal that was not achieved is extended to a set with a goal that is reached. This can also be further extended to a set of more goals that can be achieved in the terminating state of the training sample. If the goal was to move an object to point x, but it was pushed to point y, the replay buffer would use the same transitions but change the the goal we wanted to achieve to y. So when replaying the same experience, the agent would be successful and earn an useful reward. This does not help the agent learn how to reach the goal it wanted to reach initially, but it learns how to reach other goals. Being able to reach those other already achieved goals might be beneficial in learning how to reach the goal it actually wanted to achieve. HER is mainly used for tasks with multiple goals, but it was shown that it also improves the training of tasks with only a single goal. [3] Interestingly, Andrychowicz et al. have shown HER performs has problems when using shaped rewards. [3]

3. Methodology

3.1. Simulation Environment

This section describes the environment and tools used for the experiments.

MuJoCo is a physics engine used to simulate the physical models of the environment. MuJoCo is currently still in development and is improved, so there are many different versions. For this thesis, MuJoCo 2.0 for Linux (Ubuntu 16.04) is used.
mujoco-py is used as an interface to allow the usage of MuJoCo in python scripts. mujoco-py is required for OpenAI Gym to work.
OpenAI developed OpenAI Gym, a toolkit to create and use environments, and use these environments to test and compare algorithms for RL. OpenAI Gym only provides the environment part of RL, the agent has to be written by the user or by using OpenAI baselines. The robotic environments require MuJoCo. In our case, we will create our own environments and compare them to the existing ones.
OpenAI baselines is a toolkit with high-quality implementations of RL algorithms. It supplements the OpenAI Gym toolkit. For each of the environments by OpenAI Gym, any OpenAI baselines RL algorithm can be used. In our case, we will focus mainly on HER.
All the experiments will run on a 12 core machine using 10 cores. The machine is running on the operating system Linux Ubuntu 16.04. The parameters used are listed in the appendix. If not otherwise specified, each experiment will be run for 50 episodes. which takes about 2 hours of training time each.

3.1.1. MuJoCo

MuJoCo stands for ”Multi-Joint dynamics with Contact”. It is a physics engine for model based control and was developed by Todorov et al. [24]. MuJoCo was developed for research in areas with fast and accurate simulation, like robotics. As its name suggests, multi-joint dynamics and contact responses and contact responses are a main focus of the engine. They represented multi-joint dynamics in generalized coordinates and computed them with recursive algorithms. For the contact responses, they wrote algorithms based on a modern velocity-stepping approach. MuJoCo was developed to be fast and accurate, especially for computationally intensive processes, which are common in simulations of physics.

Todorov et al. compared MuJoCo to SD/FAST, another tool to simulate physics of mechanical systems. Even though SD/FAST uses model-specific code, which was expected to be much fast, MuJoCo was quite comparable. Their tests with a 12-core machine showed 400.000 dynamics evaluations per second for a 3D humanoid with 18 DOF and 6 active contacts. Creating models for MuJoCo is quite simple. For MuJoCo, XML files can be used, which are simple to understand and provide high transparency.
To use MuJoCo with Python, mujoco-py was created by OpenAI. Mujoco-py currently supports compatibility of MuJoCo with Python 3.

3.1.2. OpenAI

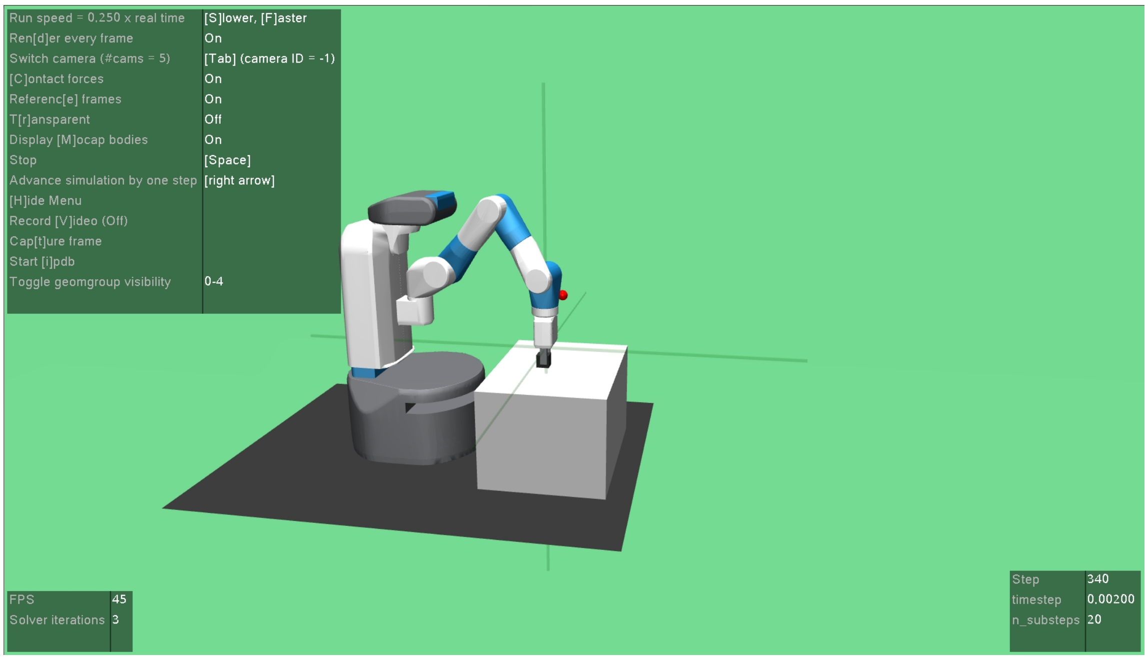
Brockman et al. developed OpenAI Gym [6] and Dhariwal et al. developed OpenAI Baselines [8]. Both are freely accessible on Github.

OpenAI Gym

OpenAI Gym provides an environment to test a RL algorithm with. OpenAI Gym already contains many environments to use, like Atari games, classic control problems and robotics. In this thesis, the robotics environment will be used, as it provides four environments with fetch robots that use a robotic arm. But OpenAI also allows the user to create his own environments. For the experiments, a few more robotics environments will be created. For the robotics environments MuJoCo is required. The agent and the RL algorithms that are required for RL to interact with the environment have to be either coded by the user or provided by OpenAI Baselines.

OpenAI Baselines

OpenAI Baselines provides a set of high-quality implementations of RL algorithms. It can be used together with OpenAI Gym. Provided algorithms contain Advantage Actor Critic, Actor critic with experience replay, Actor Critic using Kronecker-Factored Trust Region, DDPG, Deep Q-Networks, Generative Adversarial Imitation Learning, Proximal Policy Optimization, Trust Region Policy Optimization and HER. For our purposes, HER in conjunction with DDPG will be used. OpenAI Baselines also requires Tensorflow to work.



Run speed = 0.250 x real time	[S]lower, [F]aster
Ren[d]er every frame	On
Switch camera (#cams = 5)	[Tab] (camera ID = -1)
[C]ontact forces	On
Referenc[e] frames	On
T[r]ansparent	Off
Display [M]ocap bodies	On
Stop	[Space]
Advance simulation by one step	[right arrow]
[H]ide Menu	
Record [V]ideo (Off)	
Cap[t]ure frame	
Start [i]pdb	
Toggle geomgroup visibility	0-4

FPS	45
Solver iterations	3

Step	340
timestep	0.00200
n_substeps	20

Figure 3.1.: the FetchPickAndPlace-v1 environment by OpenAI Gym run with mujoco-py

3.2. Model

The environment model used is quite simple. The environments are the basic fetch environments in the robotics environment package of OpenAI gym [17], for the extended harder experiments, the object is changed to a ball. Otherwise, only positions of the object and goal as well as properties like size and friction are changed.

In figure 3.1 the environment for the task ”FetchPickAndPlace” by OpenAI is shown. A fetch robotic arm with 7 DOF is used. Its end effector is a two-fingered parallel gripper. Even though it is simulated in MuJoCo, Andrychowicz et al. have shown in their paper on HER, that the robotic arm also performed well in real life without any finetuning. [3]

All the tasks require an object (either a cube or a ball) to be moved to a goal that is presented by a red point. Also a table is used to increase the height of the object and goal, so the fetch robot can grab the object. Some parameters except from position parameters that might differ depending on the robotics environment are listed here.

- has_object (boolean): describes whether the environment contains an object or not
- block_gripper (boolean): if True, the fetch robot can not open or close its gripper

- n_substeps (integer): number of timesteps before the next action is chosen
- target_in_the_air (boolean): if True, the target is is not on the ground or table
- object_range (float): defines a range in which the object will be randomly and uniformly placed
- target_range (float): defines the range in which the goal will be randomly and uniformly placed
- distance_threshold (float): the distance the object can be to the goal for the goal to be still successful

In RL, the agent chooses an action and receives a reward and an observation after each step. actions, rewards and observations are handled in OpenAI Gym as follows.

- Action: The action space is defined as a 4-dimensional Box space, which is an array of 4 floats, which are continuous and between -1 and 1. The 4 floats define how to change the x-position, y-position and z-position of the gripper and how much to open/close the gripper (if the parameter block_gripper is False)
- Reward: The reward defined as a float. In the robotics environments, the agent receives a reward of -1.0 in each timestep, in which the goal is not reached.
- Observation: The observation space contains 3 parts: the achieved_goal, defined by a 3-dimensional Box space, which is useful for HER. the desired_goal, also defined by a 3-dimensional Box space. the observation: defined by a 25-dimnsional Box space

If not stated otherwise, each environment sample will be run for 50 steps. With the number of substeps being usually fixed to 20, each sample will run for 1000 timesteps.

4. Experiments

This chapter describes the experiments that were carried out and their results. This chapter is divided into 3 subchapters.
First the four robotics environments ″FetchReach″, ″FetchPush″, ″FetchSlide″ and ″FetchPickAndPlace″ that are already integrated in OpenAI Gym, will be compared and used as benchmarks. Then two self created environments, FetchSlideball and FetchToss will be described.

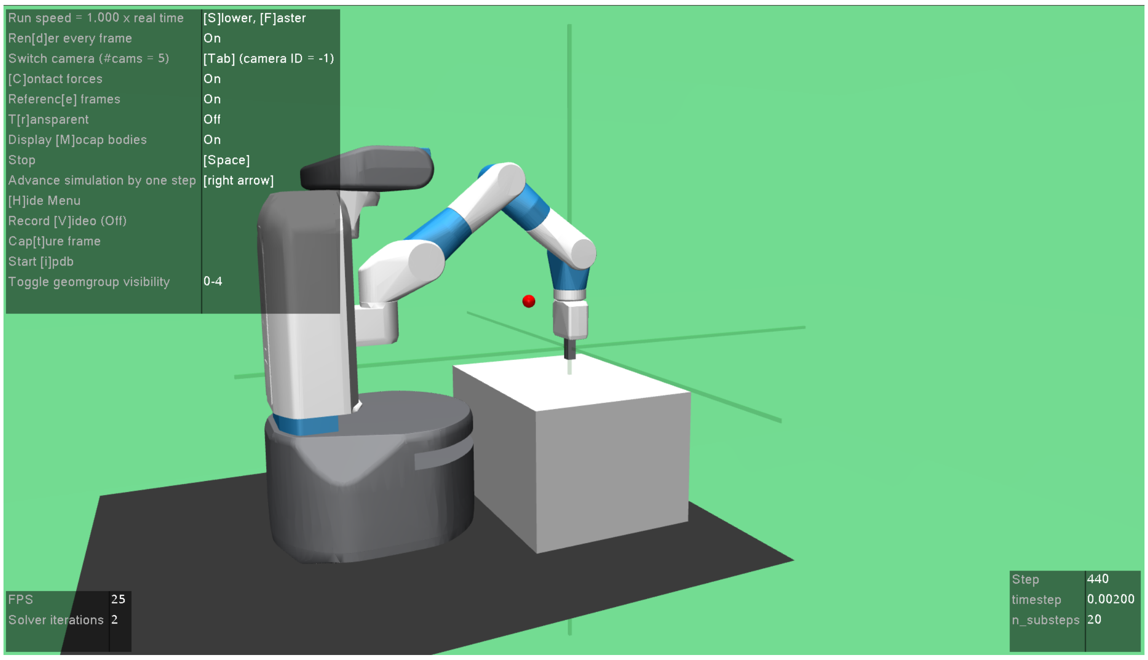
″FetchSlideball″ is an extension of FetchSlide. We changed the object from a cylinder to a ball and increased the distance to the goal. This will be compared with the FetchSlide environment of the benchmarks. It is planned to improve this environment to an environment that simulates a golf course in the future.

″FetchToss″ requires the agent to toss an object to a goal that is outside of the agents reach. It requires the agent to grab the object and then find the right trajectory to move the object and release it to toss it. The first part of the task is comparable to FetchPickAndPlace that requires the fetch robot to fetch the object. It was planned to improve this environment to make it toss a ball into a basket like in basketball.

In each chapter, the tasks and environment will be described. The action space, observation space and rewards to control the agent are also described. Then the results are discussed and compared to other tasks.

4.1. Fetch environments by OpenAI

In this section, the four basic robotics environments of OpenAI with the fetch robotic arm are shortly described and compared.



```
Run speed = 1.000 x real time    [S]lower, [F]aster
Ren[d]er every frame             On
Switch camera (#cams = 5)        [Tab] (camera ID = -1)
[C]ontact forces                 On
Referenc[e] frames               On
T[r]ansparent                    Off
Display [M]ocap bodies           On
Stop                             [Space]
Advance simulation by one step   [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility      0-4
```

```
FPS                25
Solver iterations  2
```

```
Step         440
timestep     0.00200
n_substeps   20
```

Figure 4.1.: FetchReach-v1

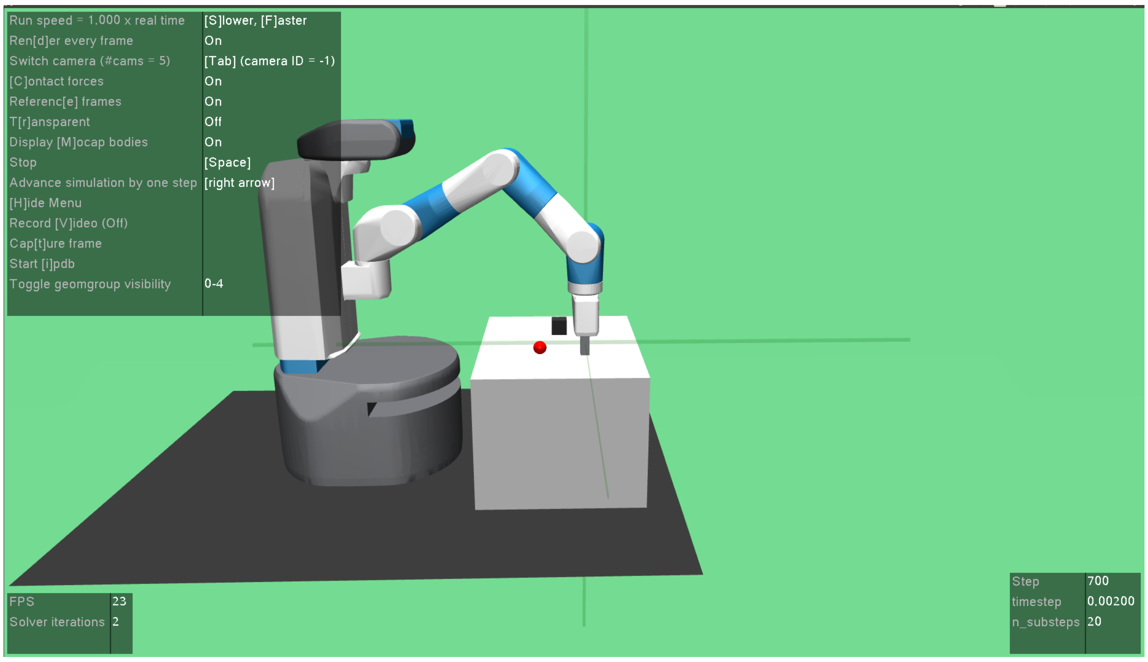
4.1.1. FetchReach

The environment FetchReach is the simplest of the OpenAI robotic environments. As can be seen in figure 4.1, the environment consists of the fetch robot, a table and a red ball indicating the goal. The task is to make the robot move its gripper to the same position as the goal. The goal can be on the table as well as in the air. Also, the goal is only above the table, so the robot is always able to reach the goal. The only thing the robot has to figure out is a path from one starting point to different points.

Figure 4.5 shows how effective HER is. After just about 7 epochs, the success rate is already at 100%.

4.1.2. FetchPush

FetchPush is already much harder than FetchReach. The environment is the same as in FetchReach, but an object in form of a cube was added. This can be seen in figure 4.2. The goal is to move the cube to the goal position. This requires the robot to learn how to move its gripper from the start position to the side of the cube that is away from the goal. Then it needs to move its gripper towards the goal to solve the task. Still, HER proves to be



```
Run speed = 1.000 x real time    [S]lower, [F]aster
Ren[d]er every frame             On
Switch camera (#cams = 5)        [Tab] (camera ID = -1)
[C]ontact forces                 On
Referenc[e] frames               On
T[r]ansparent                    Off
Display [M]ocap bodies           On
Stop                             [Space]
Advance simulation by one step   [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility      0-4

FPS                 23
Solver iterations   2

Step         700
timestep     0.00200
n_substeps   20
```

Figure 4.2.: FetchPush-v1

quite powerful. After about 14 epochs, In comparison to FetchReach, it took about twice the time to learn to solve the task.

4.1.3. FetchSlide

The FetchSlide environment is quite similar to FetchPush. The task is the same, the robot has to move an object, this time it is a cylinder (similar to a curling stone for curling). There are two main differences to FetchPush which make the task harder. The cylinder has less friction and slides, so the robot needs to carefully move the cylinder to avoid making it slide too far. Also the goal position is further away, partly even outside of the robotic arms' range. So the sliding property of the cylinder has to be used in order to reach those goals.

Using HER provides worse results than FetchPush. After training, the robot only reaches a success rate of about 60%. In the failed attempts the robot learned to push the cylinder in the right direction, only the distance is not right. The cylinder either slides too far or does not slide far enough. Most of these fails have goal positions that are outside of the robotic arms' range. Interestingly, the robot also struggles with goals that are inside of the robotic arms' range. This is probably due to the sliding property. When touching

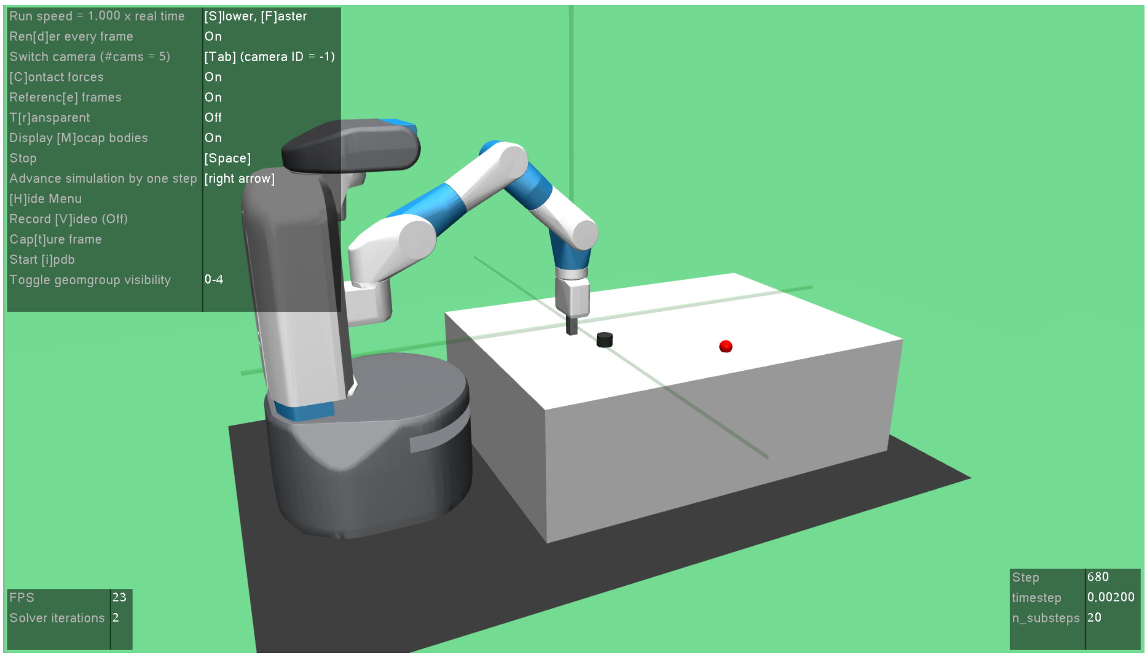
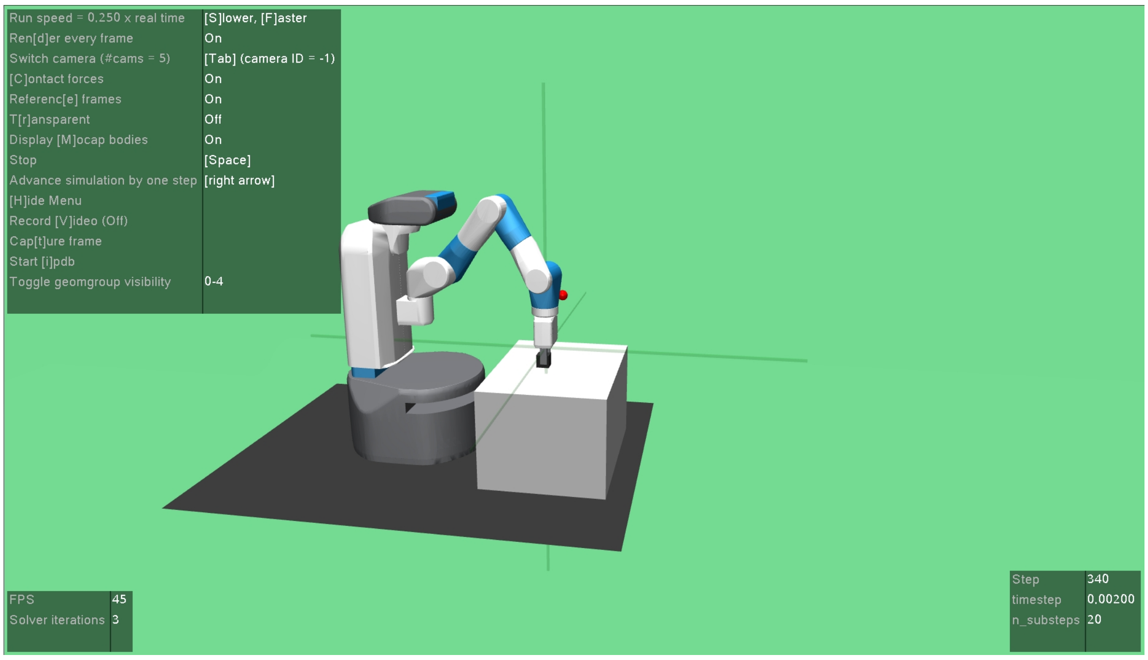


Figure 4.3.: FetchSlide-v1

the cylinder while training, the cylinder will probably slide further away and might often land outside of the robotic arms' reach. With HER, the agent learns to reach the states that it already reached at some point. Because the cylinder has a lot more positions it can be in due to its sliding property, other than in FetchPush, it does not learn to move the cylinder inside its range as well as in FetchPush.

4.1.4. FetchPickAndPlace

The environment for FetchPickAndPlace is exactly the same as in FetchPush except for the goal position. It can also be in the air. This requires the robot to use its gripper to fetch the cube and move it to the goals in the air. So the robot has to learn how to move its gripper towards the cube and open its gripper, then close its gripper to grab the cube. Then it has to move the cube to the goal position without dropping the cube by opening its gripper. The training results in figure 4.5 show that learning to solve this task works quite well with HER. After about 30 epochs it almost reaches 100% success rate. The reason why it takes much more times than the FetchPush task can be explained when comparing both. In the FetchPush task, the agent needs to learn to move its arm to the cube on the side farther away from the goal, then move its arm towards the goal. We have seen in FetchReach that it is quite simple to learn how to move the arm from one position to another. The difficulty in FetchPush comes from figuring out how to move the object.



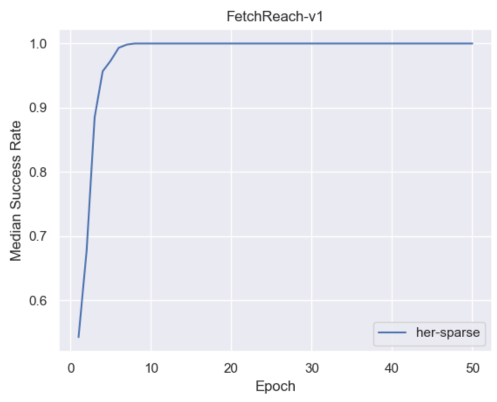
```
Run speed = 0.250 x real time     [S]lower, [F]aster
Ren[d]er every frame              On
Switch camera (#cams = 5)         [Tab] (camera ID = -1)
[C]ontact forces                  On
Referenc[e] frames                On
T[r]ansparent                     Off
Display [M]ocap bodies            On
Stop                              [Space]
Advance simulation by one step    [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility       0-4

FPS                45
Solver iterations  3

Step         340
timestep     0.00200
n_substeps   20
```

Figure 4.4.: FetchPickAndPlace-v1

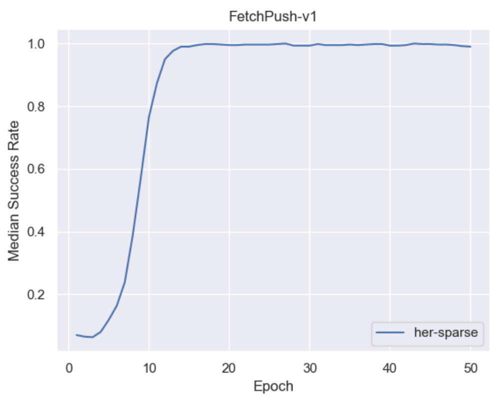
In FetchPickAndPlace this is even harder, because the opening and closing the gripper is also part of the actions it can take. Learning how to grab the cube and keep it grabbed seems to be a the cause to why it takes more time to learn.

4.1.5. Discussion of Results

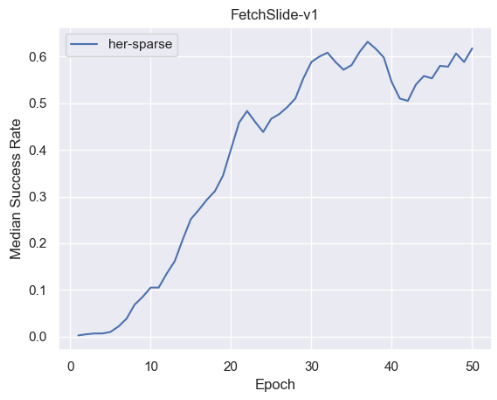
Figure 4.5 show the training results for each task. FetchReach showed that just moving the robotic arm between two points is quite fast and easy to learn.
Obviously when comparing, the harder tasks FetchSlide and FetchPickAndPlace perform worse than FetchPush and FetchReach. FetchPickAndPlace in comparison to FetchPush introduced the difficulty of having to control opening and closing the gripper. Instead of having an action space with only 3 variables like for the other tasks, this action space is extended to 4 variables, which is an extension of the action space by 33%. Having to learn how to grab the cube seems to take about 20 episodes longer than not needing to do it.
When comparing FetchSlide to FetchPush, the difference is clear. FetchSlide performs much worse.
One difference between both tasks is the control over the object. In FetchSlide it is much harder to control the cylinder while moving it. The robot either has to hit it with very precise force or stop it if the goal position is in the robots' reach. Having big fluctuations between the force used and the distance the cylinder traveled makes it hard to learn how



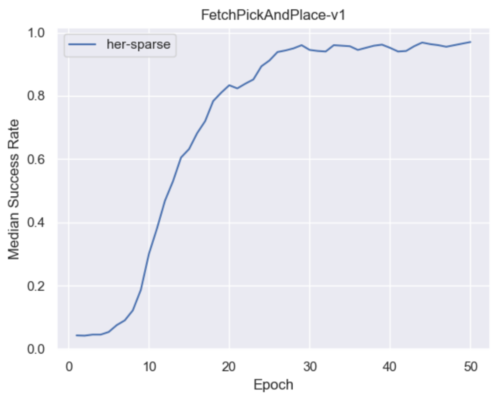
(a) FetchReach-v1



(b) FetchPush-v1



(c) FetchSlide-v1



(d) FetchPickAndPlace-v1

Figure 4.5.: Success Rate of each task after 50 episodes of training

much force exactly is needed.
Another difficulty is added by extending the range where the goal can be positioned. These multi-goal environments where the goal and object are in variable positions can be seen as a collection of many simple tasks, where each task is only about moving an object from a fixed position to another. Having a bigger goal space as increases the amount of these simple tasks greatly. As can be seen, these obstacles increase the difficulty drastically.

4.2. FetchSlideball

FetchSlideball is an extension to the environment FetchSlide. One of the future plans is to have an agent learn how to play golf. FetchSlideball made two differences to FetchSlide: the goal is put even farther in the distance and the cylinder was changed to a ball. The task will be approached in smaller steps. First a simple environment is tested where exactly the same environment as in FetchSlide is used and the only change is for the object to change from a cylinder to a ball. Through this test the difference in difficulty between using a cylinder and a ball is shown. This is needed to make FetchSlideball comparable to FetchSlide. Afterwards, for the following experiments, the friction and the steps per episode will be varied.

4.2.1. Task Description

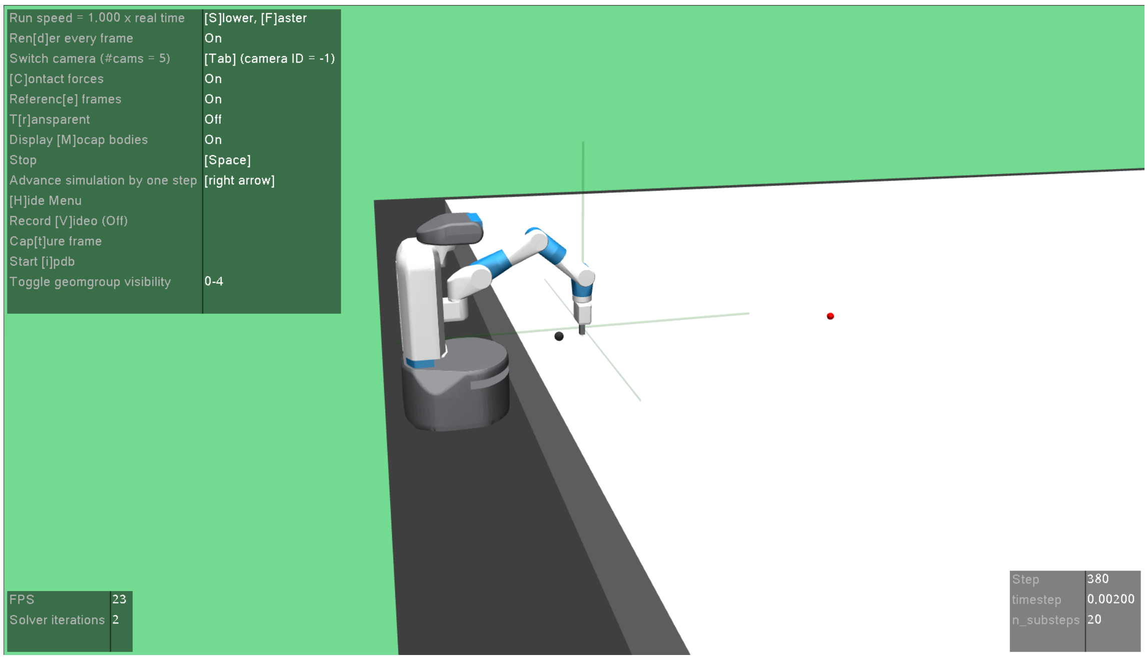
The task for FetchSlideball is exactly the same as for FetchSlide. The robotic arm has to push a ball from one position to another position, using the balls property to roll farther. Rolling the ball and sliding a cylinder might imply different friction types used, because a ball uses rolling friction instead of sliding friction which is usually much more lower. However, in this environment the same amount of friction is used for both objects. The main difference is the stability of the object. While the cylinder can fall on its side and slide different depending on where it is pushed, the ball stays stable. Also, other than in FetchSlide, the goal position is guaranteed to be outside the robots' reach. This should make it much harder for the agent to learn how to solve task.

4.2.2. Environment

The environment can be seen in figure 4.6. The size of the table was increased drastically. This was done to ensure that the goal would be on the table. The table is just much bigger than necessary to be able to accomodate future environments where the goal will be put in much farther distance. The object is a ball. As usual, there is a fetch robot and a red sphere marking the goal position.

4.2.3. Results

First the FetchSlide environment was used with the only change being a ball. As can be seen in figure 4.7, FetchSlide with a ball performs much better than vanilla FetchSlide with a cylinder. Both learning curves are quite similar. They both have a success rate curve for the first 20 epochs. After the first 20 epochs, the success rate is still rising, but visibly slower. While FetchSlide with the cylinder only reaches a success rate of 60%,

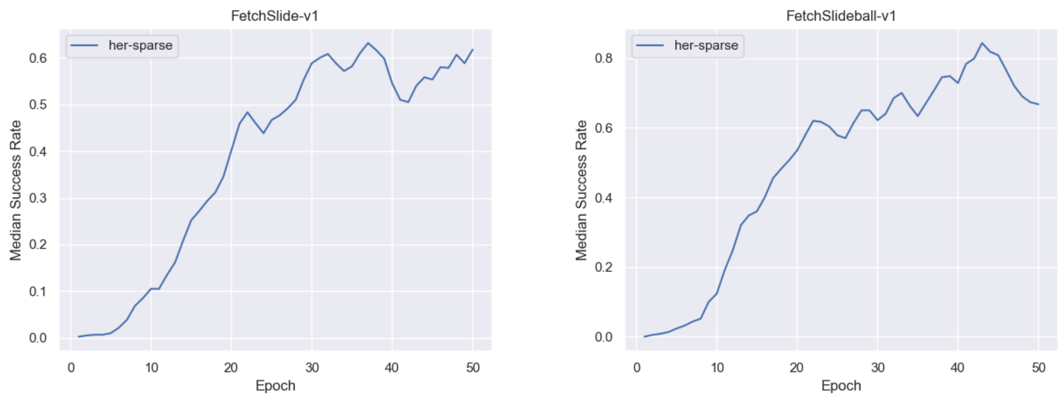


Run speed = 1.000 x real time	[S]lower, [F]aster
Ren[d]er every frame	On
Switch camera (#cams = 5)	[Tab] (camera ID = -1)
[C]ontact forces	On
Referenc[e] frames	On
T[r]ansparent	Off
Display [M]ocap bodies	On
Stop	[Space]
Advance simulation by one step	[right arrow]
[H]ide Menu	
Record [V]ideo (Off)	
Cap[t]ure frame	
Start [i]pdb	
Toggle geomgroup visibility	0-4

FPS	23
Solver iterations	2

Step	380
timestep	0.00200
n_substeps	20

Figure 4.6.: FetchSlideball-v3



(a) FetchSlide-v1 (b) FetchSlideball-v1

Figure 4.7.: FetchSlide with a cylinder (left) and with a ball (right)

FetchSlide with a ball reaches about 80%. The difference might be explained by the ball being more stable. The cylinder that is used in the normal FetchSlide environment can fall over when it is moved at a bad angle, this can not happen to a ball.
Afterwards the experiment continues for the FetchSlideball environment with a bigger

distance. The new distance from start position of the ball to the goal position is about the doubled distance of the normal FetchSlide environment. Training the FetchSlideball environment without changing any of the other parameters proved to be impossible as figure 4.8 shows. Later it was discovered to be because of a simple reason. The goal is too far away, so it is physically impossible for the robotic arm to roll the ball to the goal.

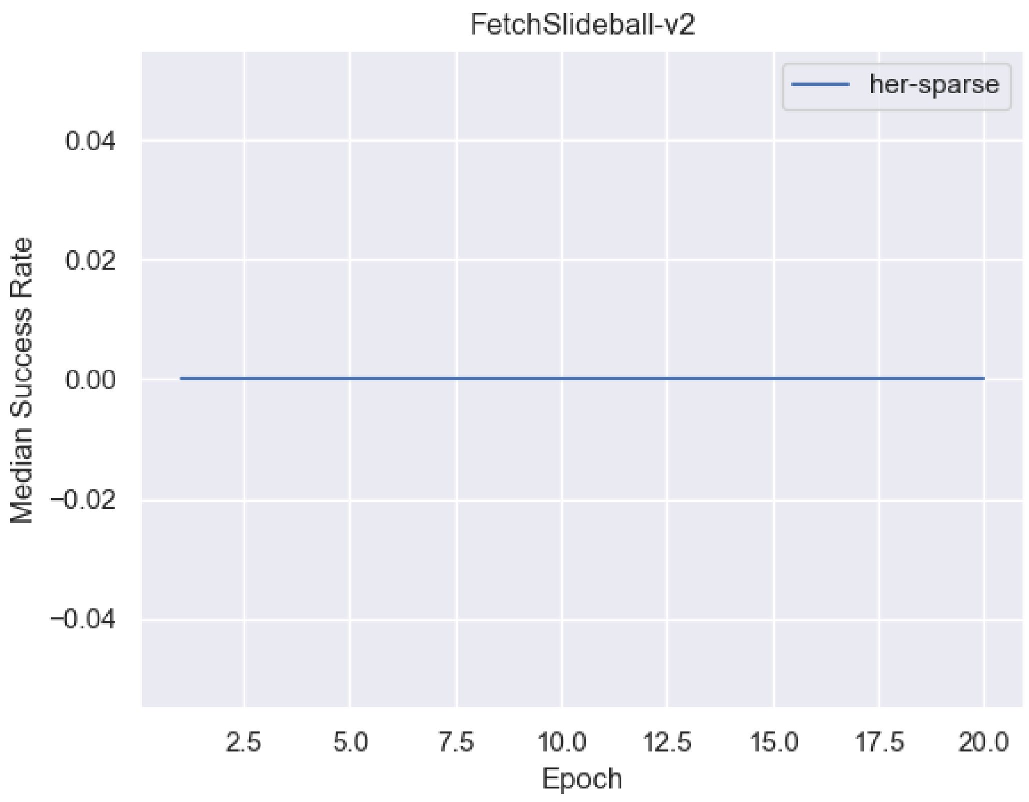


Figure 4.8.: FetchSlideball with the same friction and timestep as in FetchSlide

Reducing the friction by 50% made it barely possible to reach the goal. The goal could be reached, but the goal position is at the limit of the range that the ball could reach. Figure 4.9 showed how hard it is to learn to reach the goal. For the first 30 epochs, there was no success. Weirdly, at epochs 30 to 34 the goal was reached, but afterwards there was no success again.

Changing the balls' friction to only 10% of the original friction showed interesting results. As can be seen in figure 4.10, for the first 15 epochs there is no success, but then the success rate slowly rose. At episode 47 the success rate spiked to almost the doubled success rate. The reason behind that shows how tricky the agent can be. Each training episode takes

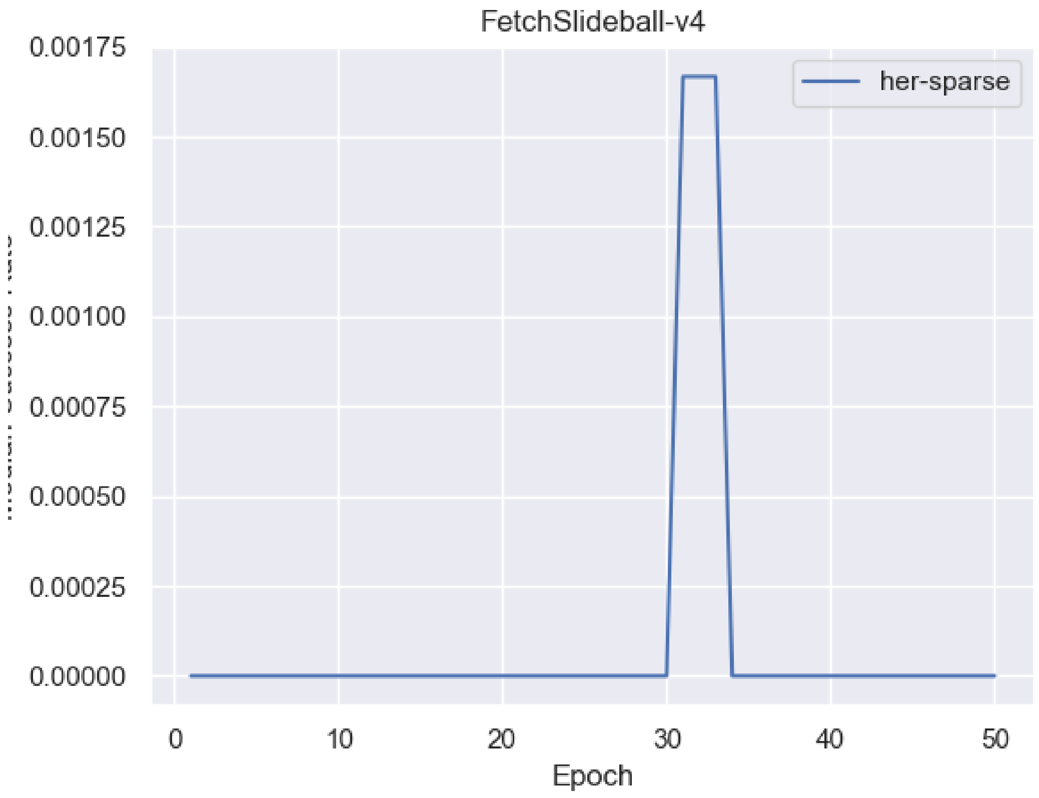


Figure 4.9.: FetchSlideball with 50% friction and more steps per episode

1000 time steps. The episode is successful when the goal is reached, to be precise, in this environment if the ball is in a close range (0,05 units of length) of the goal position in the last time step. The agent abuses this fact to solve the task different than intended. The intended solution is to roll the ball with just enough force, so that it stops at the exact goal position and stays there, so that the success condition is fulfilled and the task is successful solved. But the agent uses a different idea. It tries to hit the ball at exactly the right time, so that the ball is just at the goal position at time step 1000, the ball does not need to stop there. If the episode would take more time steps, then the ball would just roll too far, but because the episode ended at 1000 time steps, the success condition is fulfilled and the episode is counted as solved right. But even in the cases where the episode is not successful, the robotic arm slides the ball in the right direction, it just rolls too far. When using the trained policy of the 10% friction FetchSlideball environment to solve the task with 50% friction, it is getting quite close to the goal because it learned to move the ball in the right direction.

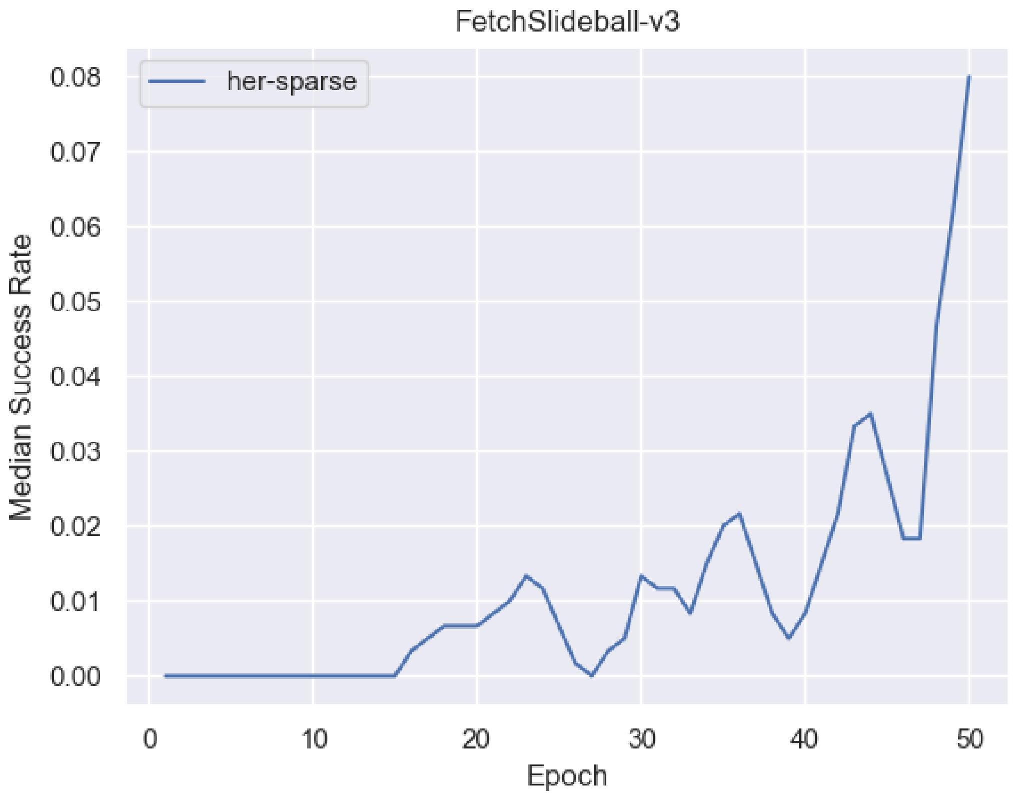


Figure 4.10.: FetchSlideball with 10% friction)

4.2.4. Discussion

Through these experiments two findings were learned. Using a ball instead of a cylinder improves the performance of the agent. This is attributed to the ball being a stable object. In this case, the ball showed an improvement of 33% over the cylinder. It might be interesting to compare the ball to other objects.
Also it was figured out that a bigger distance to the goal position increases the difficulty of the task greatly. For the FetchSlideball task with 10% friction, only 8% success rate could be reached after 50 episodes in comparison to the 60% success rate by the FetchSlide task. And even for that 8% success rate, the agent did not solve the task the intended way.
This poses two questions: Is the difficulty solely rising due the fact that the goal distance is increasing or does the difficulty also depend on the proportion between goal distance and range where the robotic arm can roll the ball to? More experiments with different friction values have to be done to answer this question.
Also, how can the agent be prevented from solving the task in an unintended way ? To

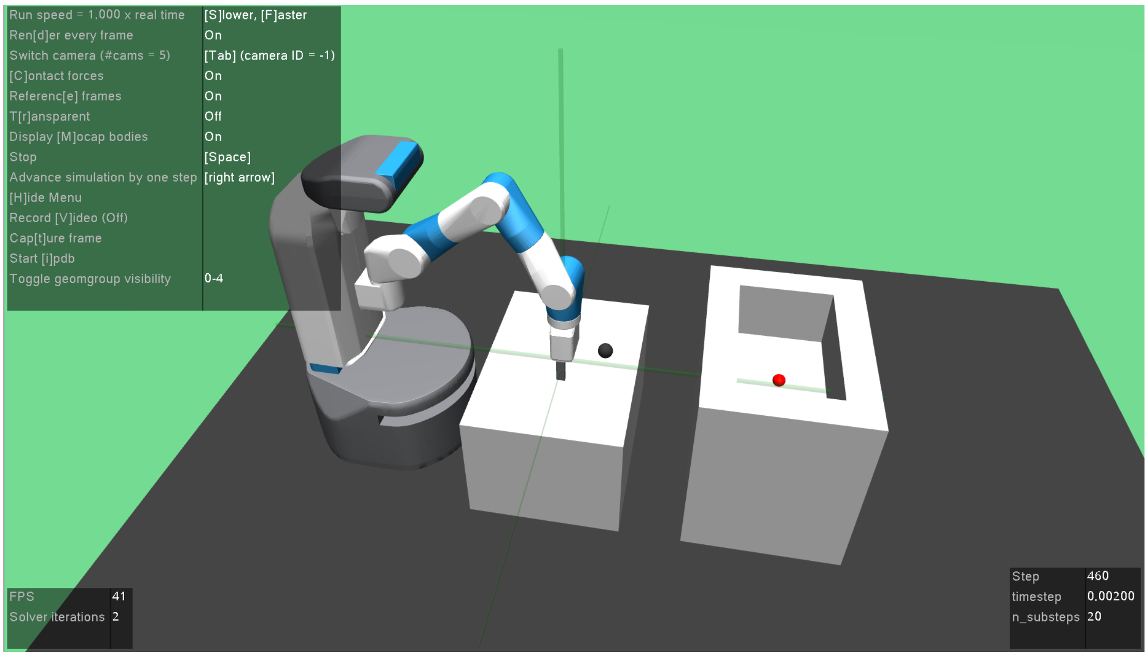
really solve the task in the intended way, the implementation of the task needs to be changed. The FetchSlideball task needs to change to have the ball lie on the goal position for some time, to count the task as successfully solved. This would prevent a ball that only touches the goal at the end to be counted.
The results have shown that vanilla HER can not solve these tasks where we increased the goal distance far outside the robotic arms' reach. To solve these tasks, further experiments with improved HER algorithms like Hindsight Goal Generation need to be done. As stated by Ren et al. [19], many hindsight experiences are not helpful to replay and therefore the hindsight goals need to be selected better to improve the performance. Especially in the case of the task FetchSlideball with 50 % friction, the goal was located at the edge of the range where the robotic arm can roll the ball to. An approach where the agent is guided to roll the ball more often in the direction of the goal would improve the performance. Further research with improved HER approaches will be done in the future.

4.3. FetchToss

FetchToss is rather different than the other environments. For future plans, FetchToss is planned to become an environment that resembles basketball. The agent should learn how to throw a ball into a basket. This environment has similarities to FetchPickAndPlace and FetchSlide, because the gripper has to be used to grab a ball and the goal is also outside of the robotic arms' range. To solve the task, we first try to change the object to a ball and see how picking a ball compares to picking a cube. Then a box is used to try to make the agent learn, how to toss the ball into the box.

4.3.1. Task Description

The task for FetchToss is to fetch a ball that is placed on the table and toss it into a box that is not reachable by the robotic arm without tossing. The goal has to be outside of reach to avoid having the robot just picking the ball up and putting it inside. The goal position and size is different than for the other tasks. For one, the goal this time is static, it will always be the same box at the same position. Also, the goal is much bigger this time. The task is fulfilled, when the ball is inside the box, it does not matter where in the box. The red sphere is just a visual mark, the actual goal is the whole box. The agent has to learn following steps: Pick up the ball like in FetchPickAndPlace, then move the object with enough force towards the goal and open the gripper to toss the ball and also hit the goal.



```
Run speed = 1.000 x real time        [S]lower, [F]aster
Ren[d]er every frame                 On
Switch camera (#cams = 5)            [Tab] (camera ID = -1)
[C]ontact forces                     On
Referenc[e] frames                   On
T[r]ansparent                        Off
Display [M]ocap bodies               On
Stop                                 [Space]
Advance simulation by one step       [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility          0-4

FPS                 41
Solver iterations   2

Step        460
timestep    0.00200
n_substeps  20
```

Figure 4.11.: FetchToss

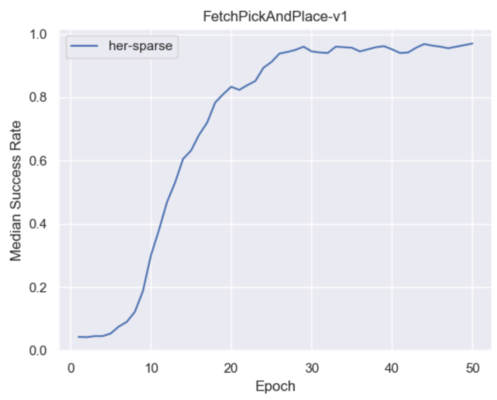
4.3.2. Environment

For this environment the environment of FetchPickAndPlace was used as a base. The object was also changed into a ball and a box was created to simulate as basket. As usual, there is also a fetch robot and a red sphere marking the goal. As mentioned, the actual goal contains the whole box, not only the position of the red sphere. Also, the goal is static.

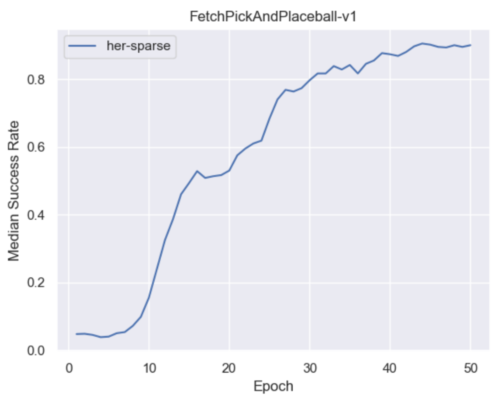
4.3.3. Results

As figure 4.12 shows, picking up a ball instead of a cube seems to perform worse. Both show similar success rate curves. FetchPickAndPlaceball seems to differ at about epoch 15. While FetchPickAndPlace still has a steep success rate curve at epoch 15, FetchPichAndPlaceball already slows down with being more successful. Overall FetchPickAndPlace with the ball shows slightly lower success rates. While it reaches about 90% success rate at 50 epochs, the vanilla FetchPickAndPlace with the cube reaches about 95% success rate.

Figure 4.13 summarizes the results for the other experiments that were run. The robotic arm does not learn how to toss the ball at all. The box was changed to a box where the



(a) FetchPickAndPlace-v1



(b) FetchPickAndPlaceball-v1

Figure 4.12.: FetchPickAndPlace with a cube(left) and a ball (right)

front is open to make it easier to toss in the back was made higher to prevent the agent from throwing the ball over the box. This also showed the same results. Another try was lengthening the time steps per episode from 1000 time steps to 2000 time steps, because it could just be impossible to solve the task as tossing takes some time. The ball was also made 100 times lighter (from 2 to 0.02 units of weight) which did not change the result. A path was planned manually to show if the reason for failing might be because the task itself is impossible. For the environment with the lighter ball a path to solve the task is possible which can be seen in Figure 4.14. This means that vanilla HER can not solve this task. Also tossing a cube instead of the ball does not work. This also proved to be unsuccessful.

4.3.4. Discussion

Picking up a cube seemed to be easier than the ball. A reason might be because of their size form. A ball with radius of 0,02 units of length, and therefore a diameter of 0,04 units of length is simply smaller than a cube with each side being 0.04 units of length long. Even though both objects have 0,04 units of length at their longest part, the ball is just smaller. Also, because of the balls form, is has to be grabbed at the middle while the cube can be grabbed at any side, it will always be 0,04 units of length long. The cube is just easier to grab and harder to drop than a ball. Experiments could be done to figure out how big the ball has to show as much success as for the cube.
Learning to toss the ball is pretty difficult as the results show. It was proven that the task is physically possible by finding a path for the lighter ball. So it seems that the task of tossing a ball is too hard to learn with standard HER.

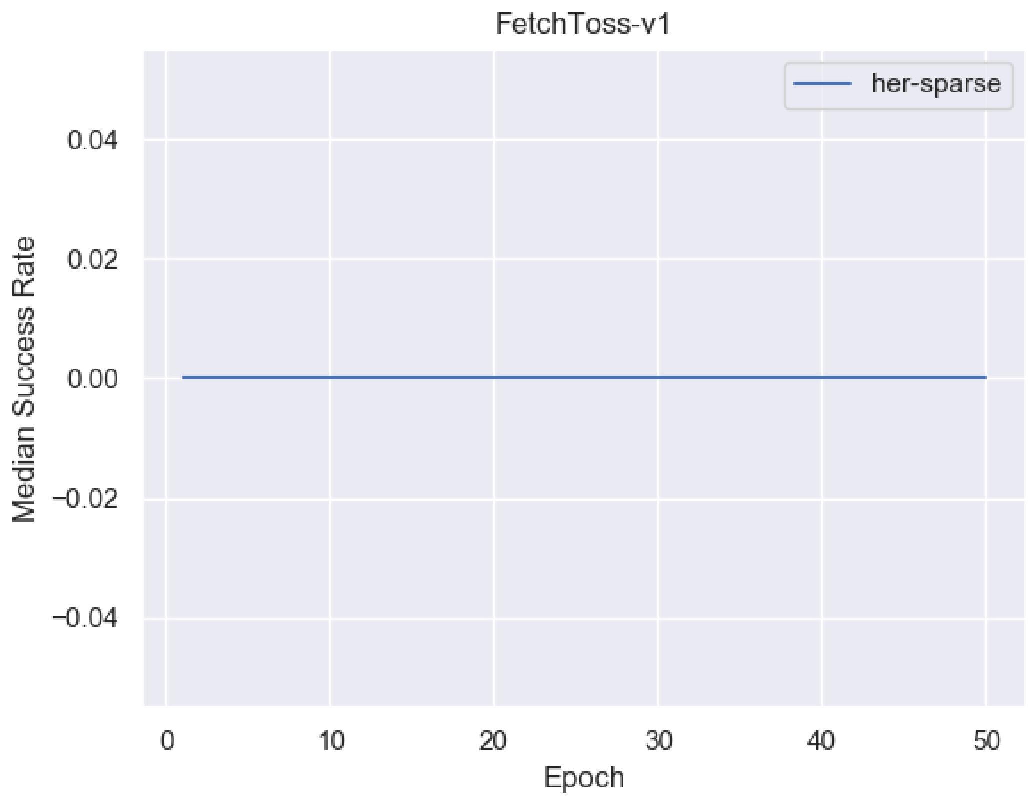
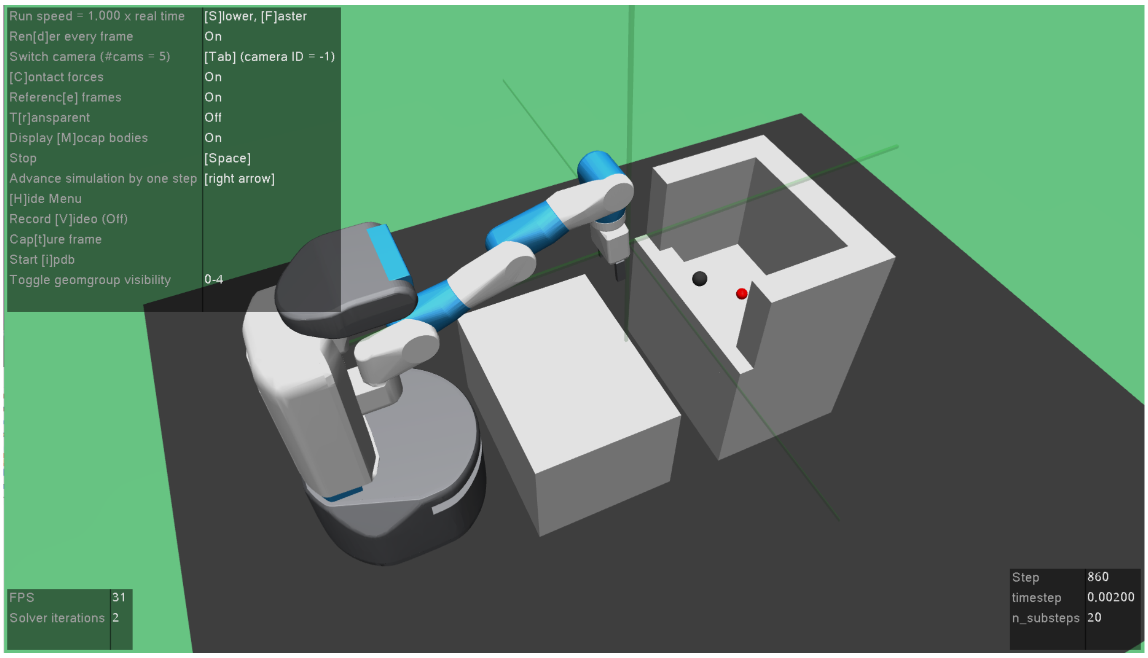


Figure 4.13.: FetchToss

As explained by Ren et al. [19], HER has the flaw that it learns how to solve goals that are equal to states that the agent already reached once, even though these goals might not be useful to learn how to solve the actual goal. In the FetchToss task learning how to toss the ball in the box seems to take too many difficult steps to learn. So vanilla HER fails. Improved HER algorithms might be useful to solve the task. Especially the energy-based hindsight experience prioritization approach by Zhao and Tresp [26] might be useful for this task because tossing requires a lot of energy and therefore prioritizing replaying experiences with high energy would be especially fitting. Further research in this direction will be done in the future.



```
Run speed = 1.000 x real time       [S]lower, [F]aster
Ren[d]er every frame                On
Switch camera (#cams = 5)           [Tab] (camera ID = -1)
[C]ontact forces                    On
Referenc[e] frames                  On
T[r]ansparent                       Off
Display [M]ocap bodies              On
Stop                                [Space]
Advance simulation by one step      [right arrow]
[H]ide Menu
Record [V]ideo (Off)
Cap[t]ure frame
Start [i]pdb
Toggle geomgroup visibility         0-4
```

```
FPS                31
Solver iterations  2
```

```
Step        860
timestep    0.00200
n_substeps  20
```

Figure 4.14.: FetchToss, with lighter ball and more time steps

5. Conclusion and Future Work

Using RL for robotic arms has been difficult due to sparse rewards in robotic arm tasks. The concept of HER enabled working with sparse rewards by making the training samples more efficient through hindsight replay. Most of the research on HER focuses on improving the performance of HER. Instead in this thesis, harder tasks were constructed to use HER on.
Two environments were built to test HER on, FetchSlideball and FetchToss.
FetchSlideball is similar to the already existing task FetchSlide which is usually used as a benchmark for HER. The task involves using the robotic arm to slide an object to a point that might be outside of the robotic arms' reach. The only difference between both environments is that FetchSlideball uses a ball instead a cylinder and the distance to the goal was increased. We first tested the FetchSlide task with a ball and compared that to the cylinder used in the default FetchSlide task. The results showed that using a ball improved the success rate for this task by 20% (from about 60% to about 80%). In a first experiment of the FetchSlideball task with the increased distance, the success rate was null. This was due to the ball not being able to roll far enough and therefore the task was impossible. To be able to reach the goal, the friction of the ball had to be modified. After halving the balls' friction the experiment was repeated. This time it was barely possible to reach the goal and solve the task, but the agent still showed almost no success. Reason for that is probably as Ren et al. stated [19], the goals that were learned through HER were not useful enough in learning how to solve the actual goal. Changing the balls friction to 10% of its original friction showed interesting results. After 50 epochs, the success rate was at 8%, which is an improvement to the former experiments. The agent learned to roll the ball in the right direction. However, instead of controlling the ball to stop at the right spot, the agent exploited the fact that a training episode only takes limited time and an episode is successful if the ball is in the goal space after the last time step. By rolling the ball with the right timing, the agent could get successful episodes by having the ball at the goal space when the episode ends - even though the ball would roll farther if the episode did not end. This showed that the agent at least learned how to roll the ball in the right direction, even for greater goal distances.
FetchToss is a much harder environment. The agent is required to use the robotic arm to fetch a ball and toss it into a box that is slightly out of the robotic arms' range. First we compared fetching a ball in contrast to fetching a cube by comparing the FetchPickAndPlace task with a ball and cube. Fetching a ball proved to be about 5% worse than fetching a cube (from about 95% to about 90%). When trying to train the agent to solve the Fetch-

Toss task, results showed no success at all. Even changing the box to make it easier to toss into, doubling the time steps for each episode, making the ball lighter or using a cube did not show any success. A path to solve the task was coded by hand which proved that it is possible to solve this task. It was just too hard for HER to solve. FetchToss requires many intermediate steps like finding the ball, picking it up, moving it in the right direction and releasing it with the right timing to toss it. Having all these steps increases the difficulty greatly. Using the energy-based experience prioritization approach by Zhao and Tresp [26] in the future might help guide the agent to learn tossing because tossing requires a high amount of energy.
This thesis showed that vanilla HER fails for both tasks FetchSlideball and FetchToss. For tasks with a high goal distance and very complicated solutions HER seemed to struggle. Future work needs to focus on solving these tasks. An obvious approach would be to use improved HER algorithms like energy-based hindsight experience prioritization to solve these tasks. Also modifying these tasks might be a possibility for research. Finding out how much different goal distances affect the success rate might be interesting. After solutions are found to make these tasks work, these tasks could be extended to a golf and basketball environment as it was intended from the start. This could include using different obstacles like walls for a golf environment. Also, other objects could be used. It would be very interesting to research tossing paper planes.

A. Simulation Parameters

_Q_lr	0.001
_action_l2	1.0
_batch_size	256
_buffer_size	1000000
_clip_obs	200.0
_hidden	256
_layers	3
_max_u	1.0
_network_class	baselines.her.actor_critic:ActorCritic
_norm_clip	5
_norm_eps	0.01
_pi_lr	0.001
_polyak	0.95
_relative_goals	False
_scope	ddpg
aux_loss_weight	0.0078
bc_loss	0
demo_batch_size	128
gamma	0.98
n_batches	40
n_test_rollouts	10
noise_eps	0.2
num_demo	100
prm_loss_weight	0.001
q_filter	0
random_eps	0.3
replay_k	4
replay_strategy	future
rollout_batch_size	2
test_with_polyak	False

List of Figures

2.1. Reinforcement Learning. An agent chooses an action to interact with the environment and gets a reward and an observation of his new state back. [5] 6
2.2. A neural network. [4] 9
2.3. A neuron and how its value is composed [18] 10

3.1. the FetchPickAndPlace-v1 environment by OpenAI Gym run with mujoco-py 15

4.1. FetchReach-v1 18
4.2. FetchPush-v1 19
4.3. FetchSlide-v1 20
4.4. FetchPickAndPlace-v1 21
4.5. Success Rate of each task after 50 episodes of training 22
4.6. FetchSlideball-v3 24
4.7. FetchSlide with a cylinder (left) and with a ball (right) 24
4.8. FetchSlideball with the same friction and timestep as in FetchSlide 25
4.9. FetchSlideball with 50% friction and more steps per episode 26
4.10. FetchSlideball with 10% friction) 27
4.11. FetchToss 29
4.12. FetchPickAndPlace with a cube(left) and a ball (right) 30
4.13. FetchToss 31
4.14. FetchToss, with lighter ball and more time steps 32

Bibliography

[1] Everything you need to know about robotic arms. accessed 27.01.2020".

[2] Software schlägt go-genie mit 4 zu 1, 2016. accessed 27.01.2020.

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2017.

[4] Faisal Awartani. Deep learning neural networks, 2018. accessed 27.01.2020.

[5] Shweta Bhatt. 5 things you need to know about reinforcement learning, 2018. accessed 27.01.2020.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[7] Wikipedia contributors. Reinforcement learning — Wikipedia, the free encyclopedia, 2020. accessed 29.01.2020".

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[9] Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12602–12613. Curran Associates, Inc., 2019.

[10] Tom Harris. The robotic arm, 2002. accessed 27.01.2020.

[11] Stefan Klanke, Dmitry Lebedev, Robert Haschke, Jochen Steil, and Helge Ritter. Dynamic path planning for a 7-dof robot arm. pages 3879 – 3884, 11 2006.

[12] Rajendra Koppula. Exploration vs. exploitation in reinforcement learning. accessed 12.02.2020.

[13] Sameera Lanka and Tianfu Wu. ARCHER: aggressive rewards to counter bias in hindsight experience replay. *CoRR*, abs/1809.02070, 2018.

[14] Matt Mazur. A step by step backpropagation example, 2015. accessed 12.02.2020.

[15] Megan Ray Nichols. Why was the robotic arm invented?, 2019. accessed 27.01.2020.

[16] Chris Nicholson. A beginner's guide to deep reinforcement learning, 2019. accessed 29.01.2020".

[17] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *CoRR*, abs/1802.09464, 2018.

[18] Prateek. Statistics is freaking hard: Wtf is activation function, 2017. accessed 27.01.2020.

[19] Zhizhou Ren, Kefan Dong, Yuan Zhou, Qiang Liu, and Jian Peng. Exploration via hindsight goal generation. *CoRR*, abs/1906.04279, 2019.

[20] Isha Salian. Supervize me: What's the difference between supervised, unsupervised, semi-supervised and reinforcement learning?, 2018. accessed 27.01.2020.

[21] David Silver and Demis Hassabis. Alphago zero: Starting from scratch, 2017. accessed 13.02.2020.

[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Alphazero: Shedding new light on chess, shogi, and go, 2018. accessed 12.02.2020.

[23] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org, 2014.

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

[25] Lilian Weng. A (long) peek into reinforcement learning, 2018. accessed 29.01.2020.

[26] Rui Zhao and Volker Tresp. Energy-based hindsight experience prioritization. *CoRR*, abs/1810.01363, 2018.

[27] Rui Zhao and Volker Tresp. Curiosity-driven experience prioritization via density estimation. *CoRR*, abs/1902.08039, 2019.