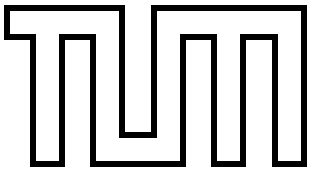


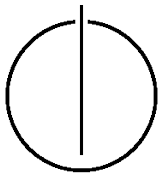
DEPARTMENT OF INFORMATICS

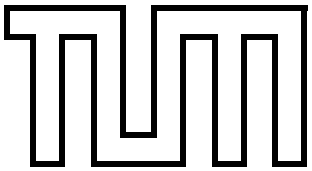
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reinforcement Learning for Path Planing of Robotic Arms

Anton Mai





DEPARTMENT OF INFORMATICS

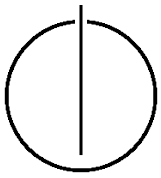
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reinforcement Learning for Path Planing of Robotic Arms

Bestrkendes Lernen zur Pfadplanung von robotischen Armen

Author:	Anton Mai
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Dr. 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, 12. Februar 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 with improvements to Hindsight Experience Replay, like Hindsight Goal Generation and Energy-Based Hindsight Experience Prioritization is required make further progress on solving these tasks.

Contents

List of Abbreviations

ATT	Address Translation Table in the Altera PCIe core
BAR	PCI Base Address Register
BSP	Board Support Package, IP stack for Altera OpenCL
CPU	Central Processing Unit
DDR	Double Data Rate, a type of memory
DMA	Direct Memory Access
DPU	Double-precision Floating Point Unit
FIFO	First-In First-Out Queue
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPC	High Performance Computing
ICD	Installable Client Driver, library that acts as a proxy between different OpenCL implementations
IOMMU	Input/Output Memory Management Unit
IP	Intellectual Property, usually refers to HDL code
IPC	Inter-Process Communication
LUT	Look-Up Table, refers to the basic FPGA building block
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
OpenCL	Open Computing Language, a popular HPC platform
OS	Operating System
PCIe	Peripheral Component Interconnect Express Bus
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
SDK	Software Development Kit
SMX	NVIDIA Streaming Multiprocessor

1. Introduction

Lee Sedol, one of the best Go players in the world, was beaten by the Go engine AlphaGo in a match. The engine was clearly stronger [1]. AlphaGo only knew the rules at the beginning 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 is reinforcement learning, an area of machine learning.

The idea of reinforcement learning is to have a state and actions that an agent can choose from. Each action results in different rewards and states. 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 got a low reward). So the next time you are in front of the goal again, you will more probably try to pass the ball. This time your teammate scored a goal (you got a high reward). From this experience you learn that it is probably better to pass the ball if you are standing in front of the goal. The concept of reinforcement learning 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.

A robotic arm resembles a human arm. It consists of segments which are connected by joints. The number of joints correspond to what is called Degrees of Freedom. A robotic arm with 5 joints would have 5 Degrees of Freedom because it can pivot in 5 ways. Each joint is connected to a step motor. Step motors make the robot move very precisely. The equivalent to a human hand is the end effector. The end effector can vary depending on the tasks.

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. This is the main reason why they are used in laboratories and hospitals for surgeries. They can also be used automatically without any human which is why they are used for manufacturing and assembly lines.

Humans still have to teach the robotic arms how to move when setting them up. For path planning of the 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. Either by testing or by using linear algebra a path can be found. A robotic arm needs 6 Degrees of Freedom to be able to move its end effector in every direction and orientation. This also means that robotic arms with more degrees of freedom 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 reinforcement learning for robotic arms is a logical approach.

...

There is an issue that prevents robotic arms to learn with reinforcement learning. It is hard to construct a suitable reward function for tasks where robotic arms are used. For example ... So either a suitable reward function has to be constructed by hand, or the simplest reward function, a binary 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 reinforcement learning for robotic arms over path planning by hand. Depending on the case it might be easier to just plan the path without reinforcement learning. Using only a sparse reward for robotic arms is as follows. a reward is given, when the goal is reached, no reward is given when the goal is not reached. Robotic arms have usually many Degrees of Freedom, so there are many actions that can be taken by 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 has been introduced. Hindsight experience replay allows a high learning rate even with sparse rewards.

Hindsight experience replay works as follows.

This thesis is structured as follows: Chapter 2 describes the theoretical background on robotic arms, reinforcement learning and algorithms like deep deterministic policy gradients and hindsight experience replay. Chapter 3 explains the methodology used for this thesis. Chapter 4 gives an overview of the simulation environment. In chapter 5, the ex-

periments 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 reinforcement learning, deep deterministic policy gradients, hindsight experience replay and hindsight goal generation are explained in this chapter.

2.1. Reinforcement Learning

Reinforcement learning is one of the main learning models of Machine learning next to Supervised Learning and Unsupervised Learning. 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. 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. 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 up. 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. In Reinforcement Learning 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. Reinforcement Learning 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. 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 Reinforcement Learning is not limited by that. The agent can improve on its own only by exploring his options. In games like Go and Chess, engines that use reinforcement learning have already far surpassed the best human players.

This section will explain the theory behind reinforcement learning. Reinforcement learning is usually modeled as a Markov Decision Process. The Markov Decision Process for Reinforcement Learning consists of following elements

- A set of states S
- A set of actions A
- The transition probability Pa(s,s') from state s to s' under action a
- The immediate reward Ra(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. 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.

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 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 V(s) is defined as follows:

The actions value function is denoted as Q(s,a) and is defined as follows:

When determining the value of a state or action, a discount factor 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. Because 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 are a set of equations that convert the value functions into the immediate and future reward:

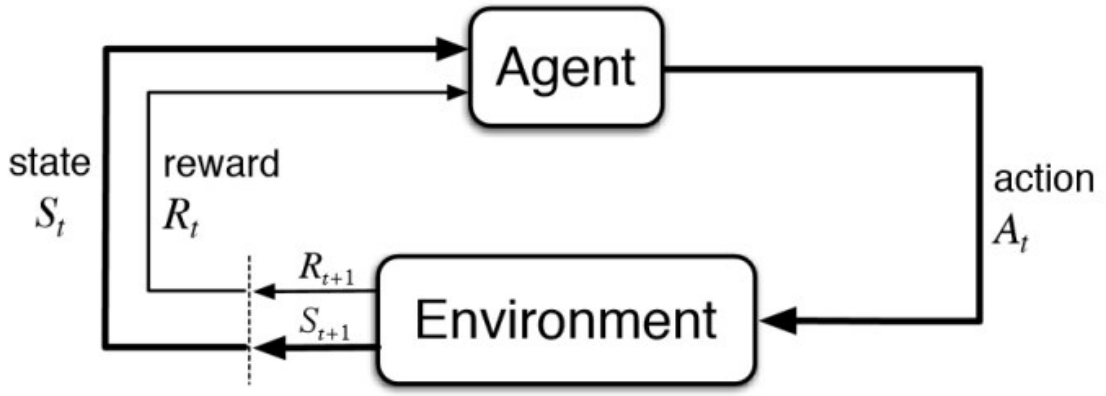


Figure 2.1.: Reinforcement Learning. An agent chooses an action to interact with the environment and gets a reward and an observation of his new state back.

The goal is to find the actions that return the maximal reward. This is displayed by the Bellman optimality equations:

To calculate the optimal values of each state and action, Dynamic Programming could be used, if we know the model fully. But even if we knew the model fully, usually the main issue is the huge state and action space, which makes it impossible to use Dynamic Programming. For reinforcement learning, neural networks can be used to approximate the value functions.

2.2. Artificial Neural Networks

Artificial Neural Networks are inspired by the human brain. The Neural Network 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 neural network 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 value of these states and actions. An example of an artificial neural network is shown in Figure 2.2.

The learning process of the neuronal network is as follows. Each neuron obtains inputs xi by the the output of the neurons in the layer before it. each input value is weighed and then added. 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

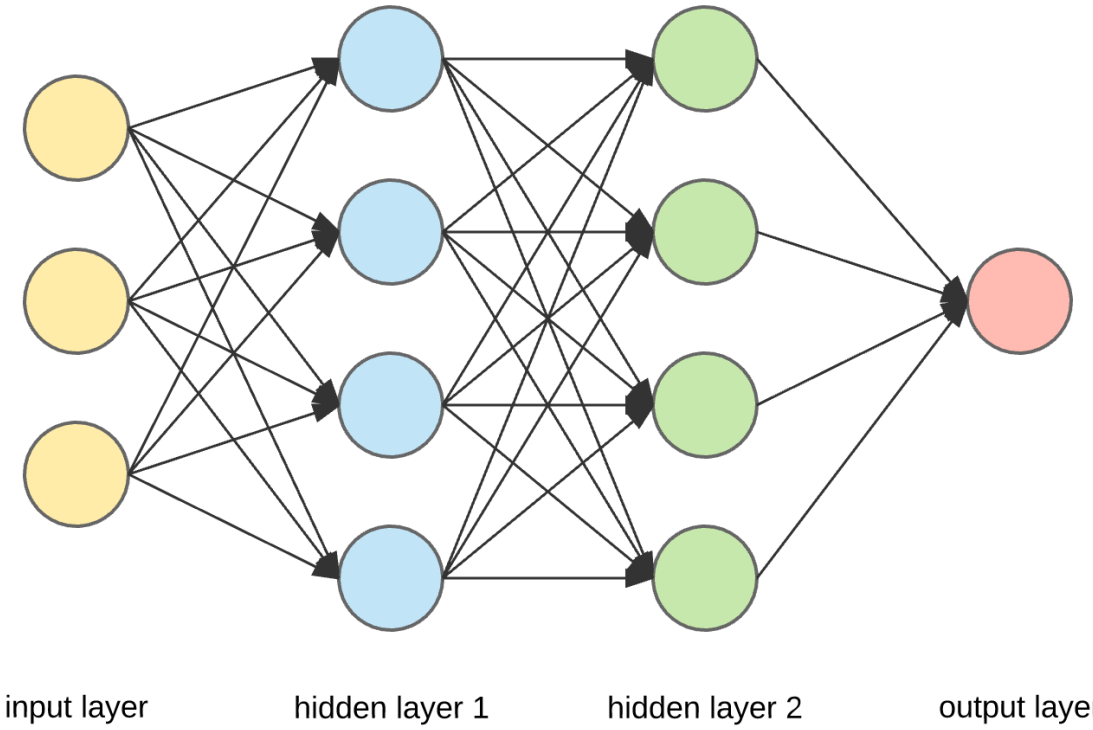


Figure 2.2.: A neural network.

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.

When the neural network outputs a value, a process called Backpropagation is used to further improve the neural network.

2.3. Deep Deterministic Policy Gradients

The algorithm Deep Deterministic Policy Gradient learns concurrently a Q-function (the action-value function) and a policy. When learning the optimal action value function Q*(s,a), the optimal action in each state can be solved by using following equation:

To solve the equations 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

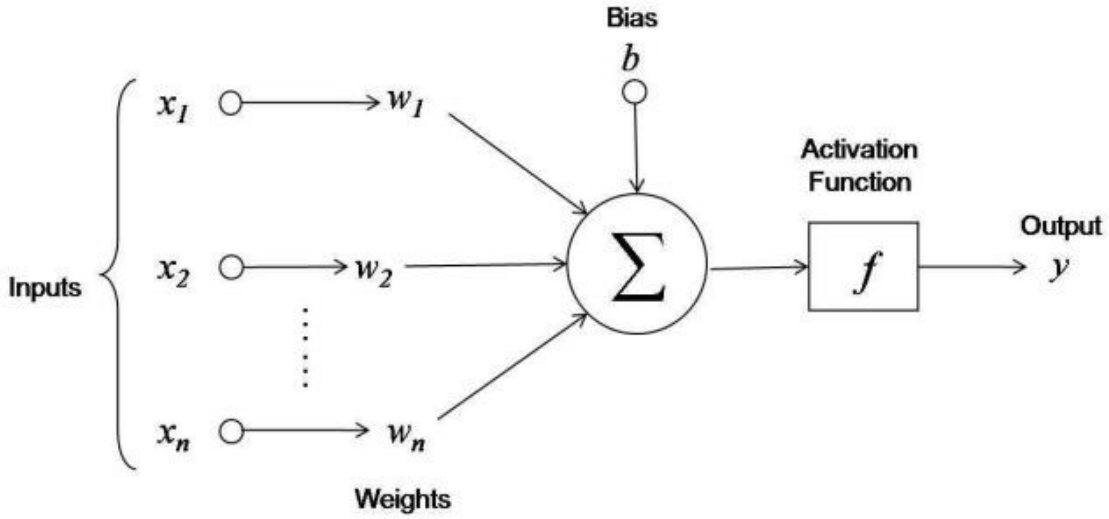


Figure 2.3.: A neuron.

space is continuous and so Q*(s,a) is expected to be differentiable in respect to the action argument. 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:

Minimizing the MSBE loss function 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 neural network might be overfitted to those experiences. 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. The following term is called target: When minimizing the MSBE loss 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 cpoy of the neural network 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 Reinforcement Learning, 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 Hindsight Experience Replay can be used to deal with this issue for robotic arms. Hindsight Experience Replay can learn efficiently from sparse rewards and can also be combined with any off-policy Reinforcement Learning algorithm. This technique is inspired by the ability of humans to learn from failures as least as much as from successes.

Hindsight Experience Replay 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 to reach other goals. Reaching those other achieved goals might be beneficial in learning how to reach the goal it actually wanted to achieve. Hindsight Experience Replay 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. Interestingly, they have shown Hindsight Experience Replay performs has problems when using shaped rewards

3. Methodology

To provide a sound answer to how hindsight experience replay performs on harder tasks in contrast to easier tasks, the obvious approach is a quantitative approach. In order to collect data, a simulation environment is built for tasks of different difficulties. The robotic arm is trained for those tasks with hindsight experience replay. The performance for the training period is measured. The data will show the performance on the tasks over time and characteristics like learning rate and consistency is shown through the data.

Data between easier and harder tasks is evaluated and compared to show how hindsight experience replay performs on different tasks. In some cases, it is clear that hindsight experience replay fails for the harder tasks. Possible reasons for the lack of performance are presented. An alternative extension for hindsight experience replay, hindsight goal generation will be used in addition to show possible approaches on solving the tasks with her for harder tasks. Hindsight goal generation will also be used on the easier tasks to make them comparable.

Validity is obviously given, because the data measured is exactly the performance and learning rate of the robotic arm which is the measurement needed to evaluate the performance of hindsight experience replay. Similar results can be reproduced when repeating the data collection. The results are not necessarily exactly the same when reproduced. This is due to the nature of reinforcement learning as there is some randomness in the Markovian decision process when choosing an action. The law of large numbers states that with rising amount of samples the results will converge towards the expected probability. In context of reinforcement learning, the training time to learn needed might vary slightly, but the end performance should converge towards the same value with rising training time.

4. 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 reinforcement learning. OpenAI Gym only provides the environment part of reinforcement learning, 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 reinforcement learning algorithms. It supplements the OpenAI Gym toolkit. For each of the environments by OpenAI Gym, any OpenAI baselines reinforcement learning algorithm can be used. In our case, we will focus mainly on Hindsight Experience Replay.

All the experiments will were done on a 12 core machine using 10 cores. The machine is running on the operating system Linux Ubuntu 16.04. The parameters used will be stated in the Model section. If not otherwise specified, each experiment will be run for 50 episodes. which takes about 2 hours training time each.

4.1. MuJoCo

MuJoCo stands for "Multi-Joint dynamics with Contact". It is a physics engine for model based control and was developed by Emanuel Todorov. 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 algorihhms based on a modern velocity-stepping approach. MuJoCo was developed to be fast and accurate, especially for computationally intensive processes, which are common in simulation of physics.

They 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 Degrees of Freedom 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 transparency.

To use MuJoCo with Python, mujoco-py was created by OpenAI. Mujoco-py currently supports compatibility of MuJoCo with Python 3.

4.2. OpenAI

OpenAI developed OpenAI Gym and OpenAI Baselines. Both are freely accessible on Github.

4.2.1. OpenAI Gym

OpenAI Gym provides an environment to test a reinforcement learning 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 reinforcement learning algorithms it uses that are required for reinforcement learning to interact with the environment has to be either written by the user or provided by OpenAI Baselines.

4.2.2. OpenAI Baselines

OpenAI Baselines provides a set of high-quality implementations of reinforcement learning 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, Deep Deterministic Policy Gradient, Deep Q-Networks, Generative Adversarial Imitation Learning, Proximal Policy Optimization, Trust Region Policy Optimization and Hindsight Experience Replay. For our purposes, Hindsight Experience Replay in conjunction with Deep Deterministic Policy Gradients will be used. OpenAI Baselines also requires Tensorflow to work.

4.3. Model

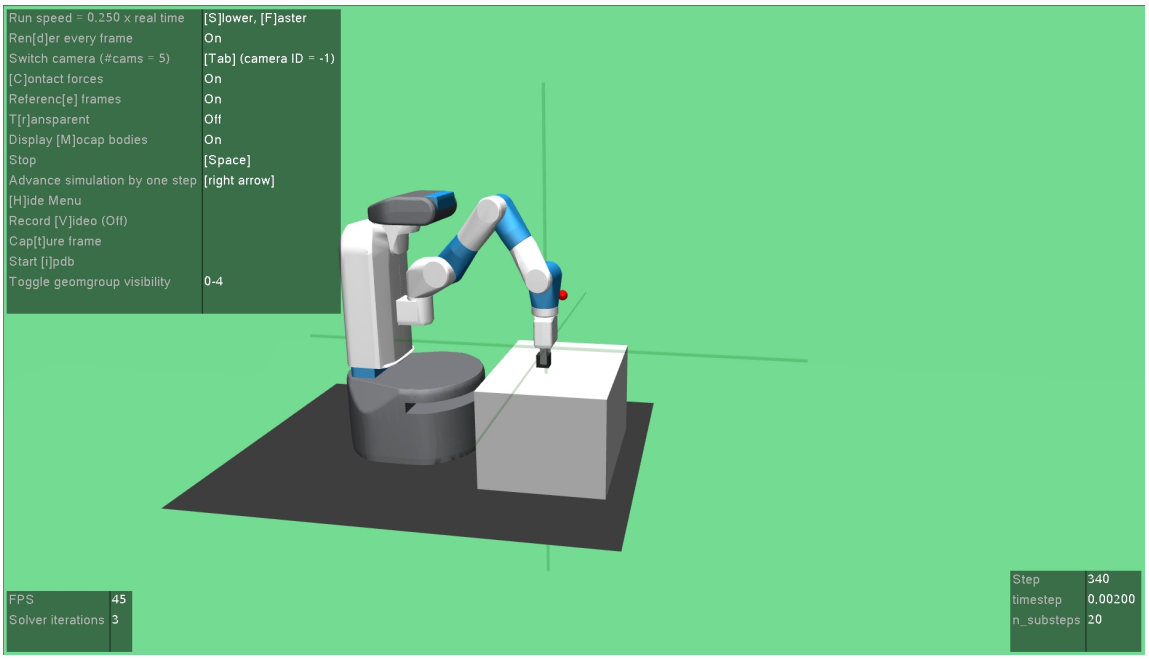


Figure 4.1.: FetchPickAndPlace-v1 Environment by OpenAI Gym run with Mujoco-py

The environment model used is quite simple. The environments are the basic models in the robotics environment package of OpenAI gym, for the extended harder experiments, only positions of the objects and goal, and properties like size and friction are changed.

In figure 4.1. the environment for the FetchPickAndPlace task of OpenAI is shown. A Fetch Robotic arm with 7 Degrees of Freedom 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 Hindsight Experience Replay, that the robotic arm also performed well in real life without any finetuning.

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 Reinforcement Learning, 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 Hindsight Experience Replay. 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 number of substeps being usually fixed to 20, each sample will run for 1000 timesteps. The hyperparameters are as follows for each of the following environments:

5. Experiments

This chapter describes the experiments made and the observations made. 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 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.

5.1. OpenAI robotics package

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

5.1.1. FetchReach

The environment FetchReach is the simplest of the OpenAI robotic environments. As can be seen in figure 5.1, the environment consists of the fetch robot, a table and a red ball

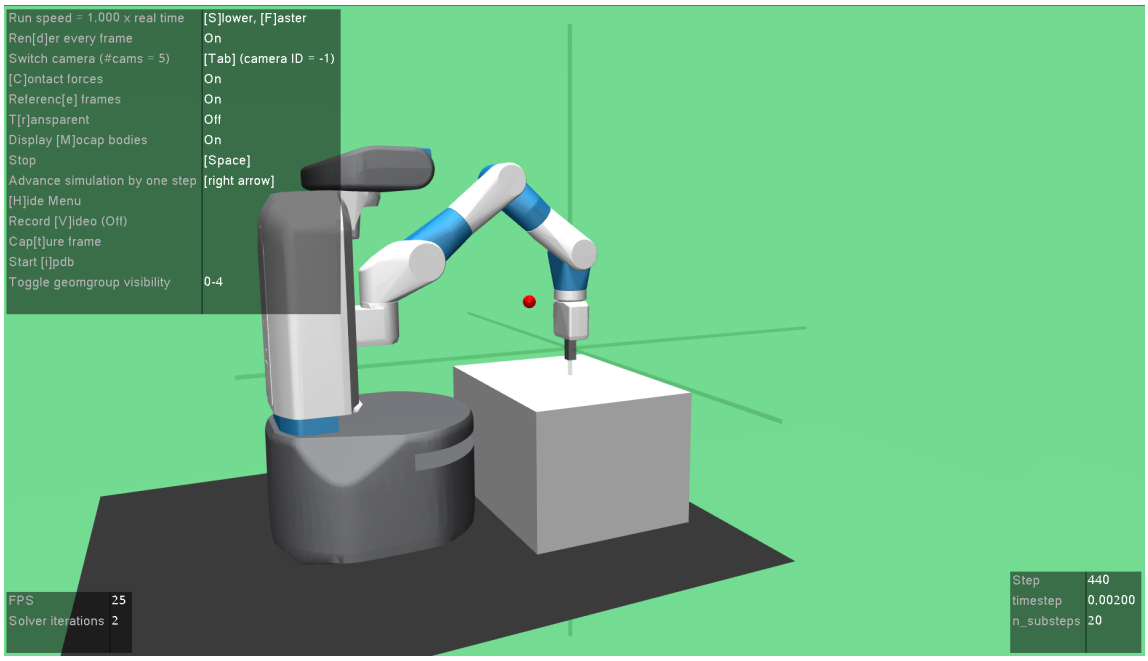


Figure 5.1.: FetchReach-v1

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 5.5 shows how effective Hindsight Experience Replay is. After just about 7 epochs, the success rate is already at 100%.

5.1.2. 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 5.2. 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, Hindsight Experience Replay proves to be quite powerful. After about 14 epochs, In comparison to FetchReach, it took about twice the time to learn to solve the task.

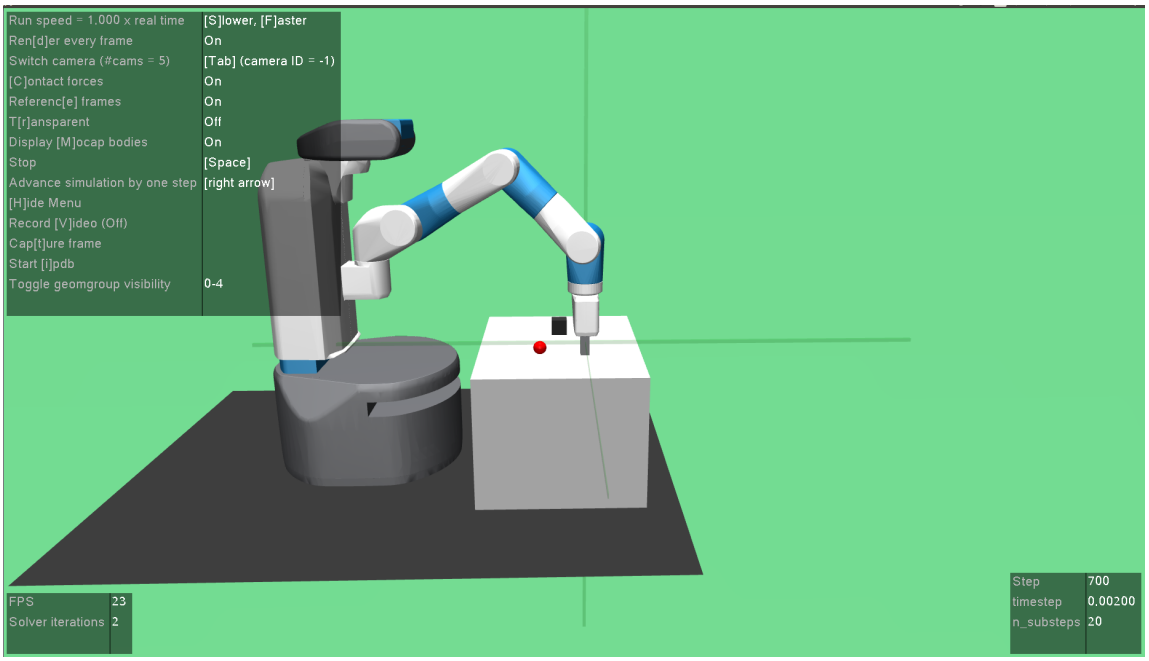


Figure 5.2.: FetchPush-v1

5.1.3. 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 Hindsight Experience Replay 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 the cylinder while training, the cylinder will probably slide further away and might often land outside of the robotic arms' reach. With Hindsight Experience Replay, 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

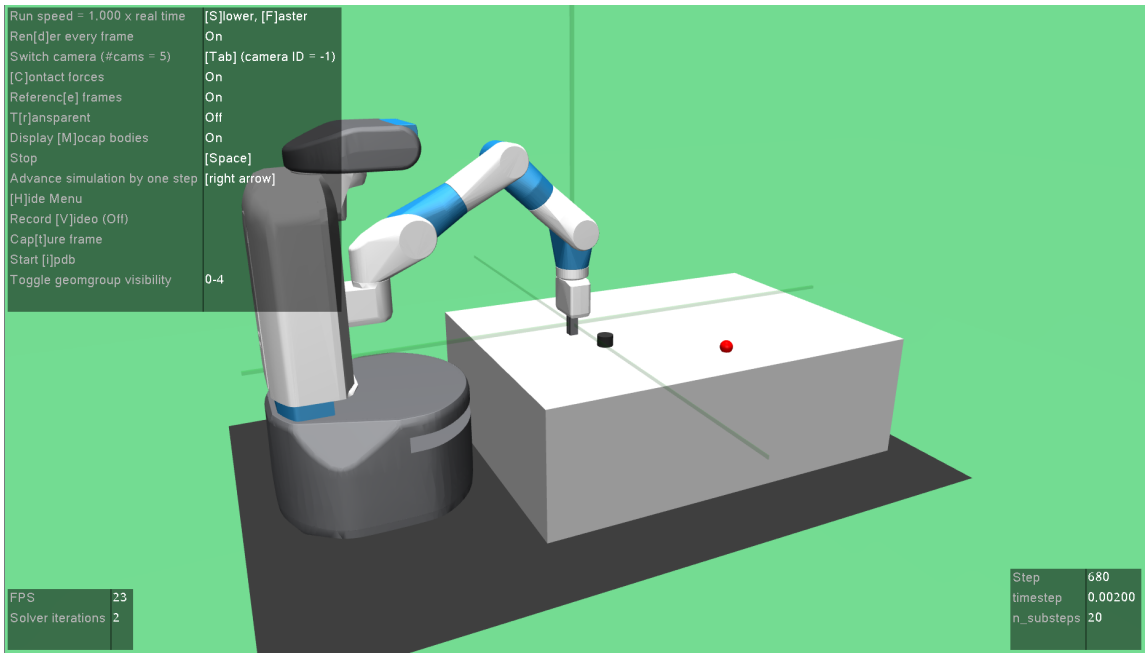


Figure 5.3.: FetchSlide-v1

than in FetchPush, it does not learn to move the cylinder inside its range as well as in FetchPush.

5.1.4. 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 5.5 show that learning to solve this task works quite well with Hindsight Experience Replay. 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. 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.

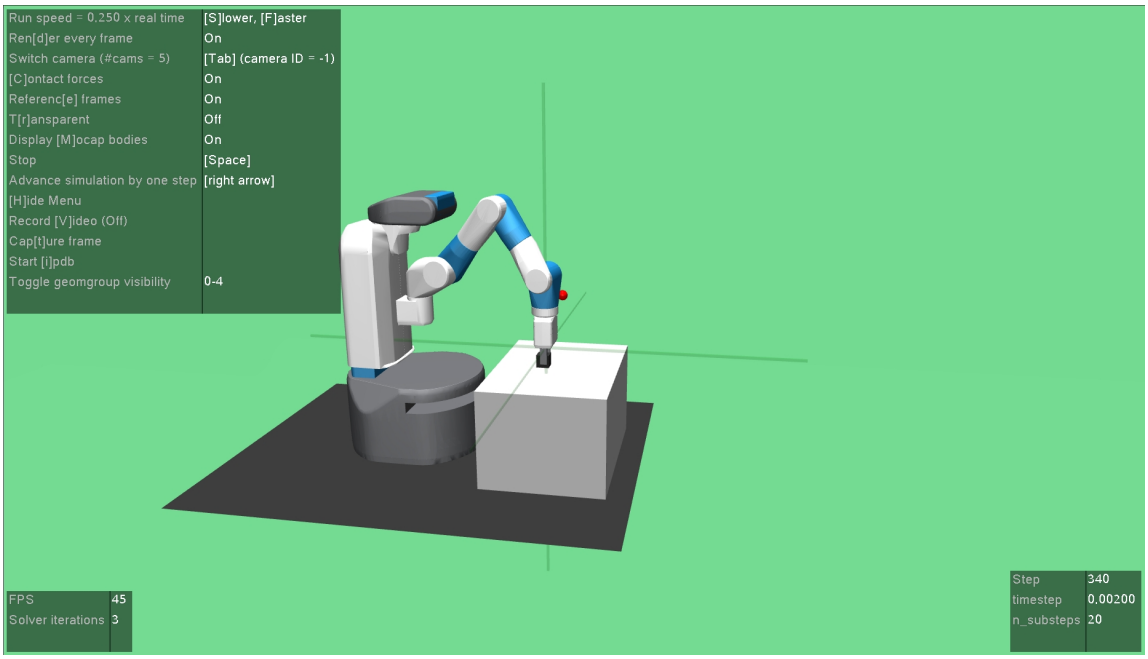


Figure 5.4.: FetchPickAndPlace-v1

5.1.5. Comparison (maybe some different section name)

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 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.

Figure 5.5.: Success Rate of each task after 50 episodes of training

5.2. FetchSlideball

FetchSlideball is an extension to the environment FetchSlide. On 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.

5.2.1. Task Description

The task for FetchSlideball is exactly the same as for FetchSlide. The robot 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, but 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.

5.2.2. Environment

The environment can be seen in figure 5.6. 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.

5.2.3. Results

First the FetchSlide environment was used with the only change being a ball. As can be seen in figure 5.7, FetchSlide with a ball performs much better than vanilla FetchSlide

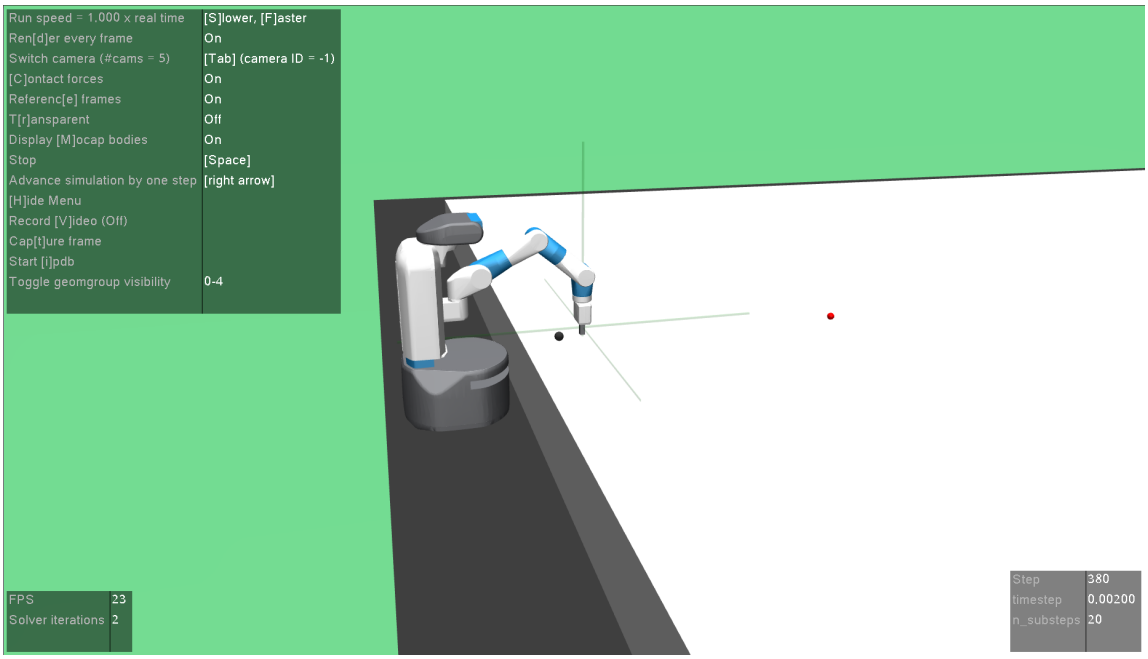


Figure 5.6.: FetchSlideball-v3

Figure 5.7.: FetchSlide with a cylinder (left) and with a ball (right)

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%, 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 parameters proved to be impossible as figure 5.8 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.

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 5.9 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.

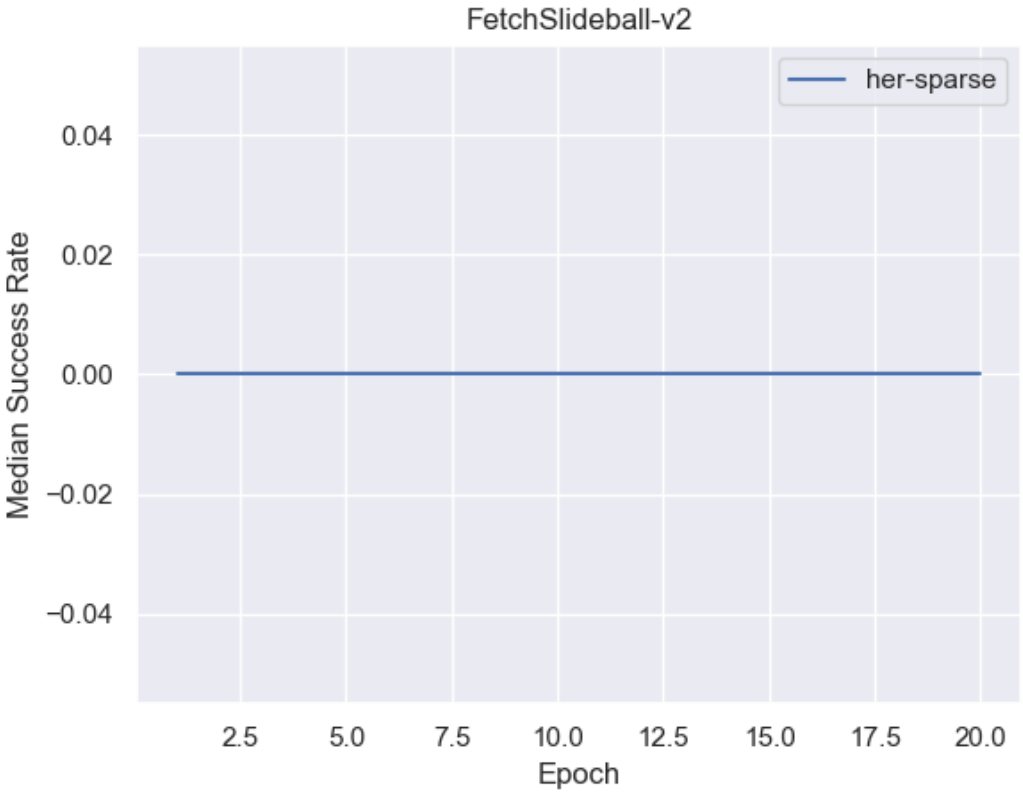


Figure 5.8.: FetchSlideball with the same friction and timestep as in FetchSlide

Changing the balls' friction to only 10% of the original friction showed interesting results. For the first 15 epochs there is no success, but then the success rate slowly rised. 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 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 timestep. 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%

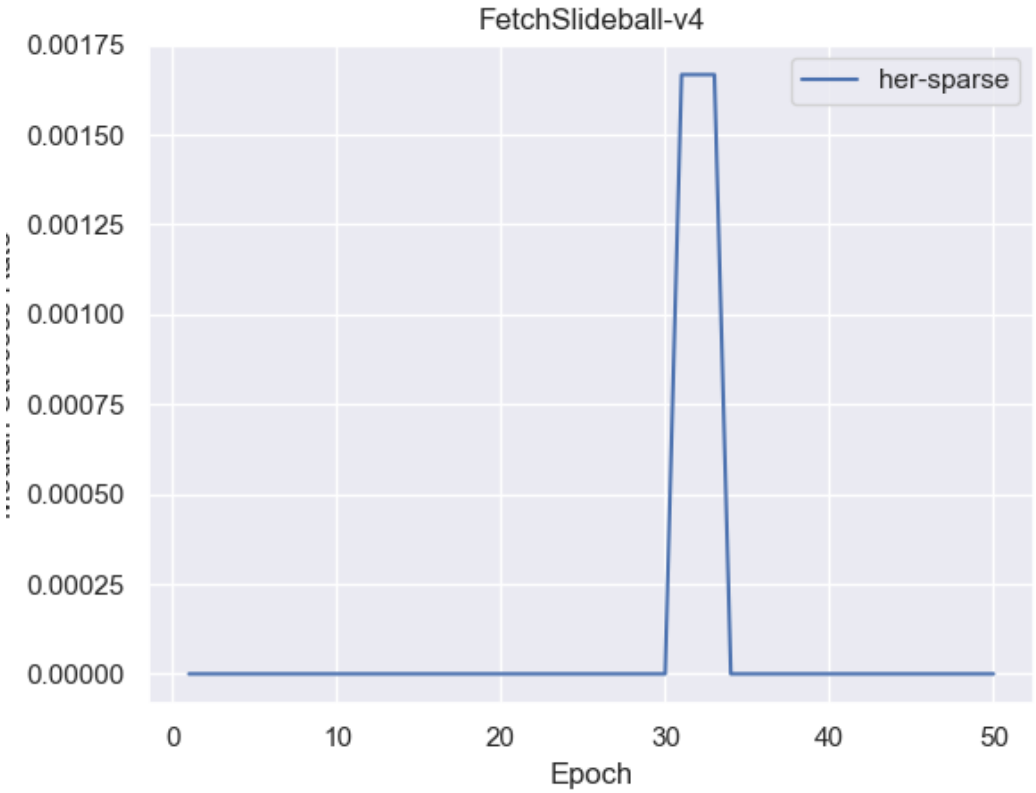


Figure 5.9.: FetchSlideball with 50% friction and more steps per episode

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.

5.2.4. 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 there some proportion between goal distance and distance that is reachable that determines the goal,

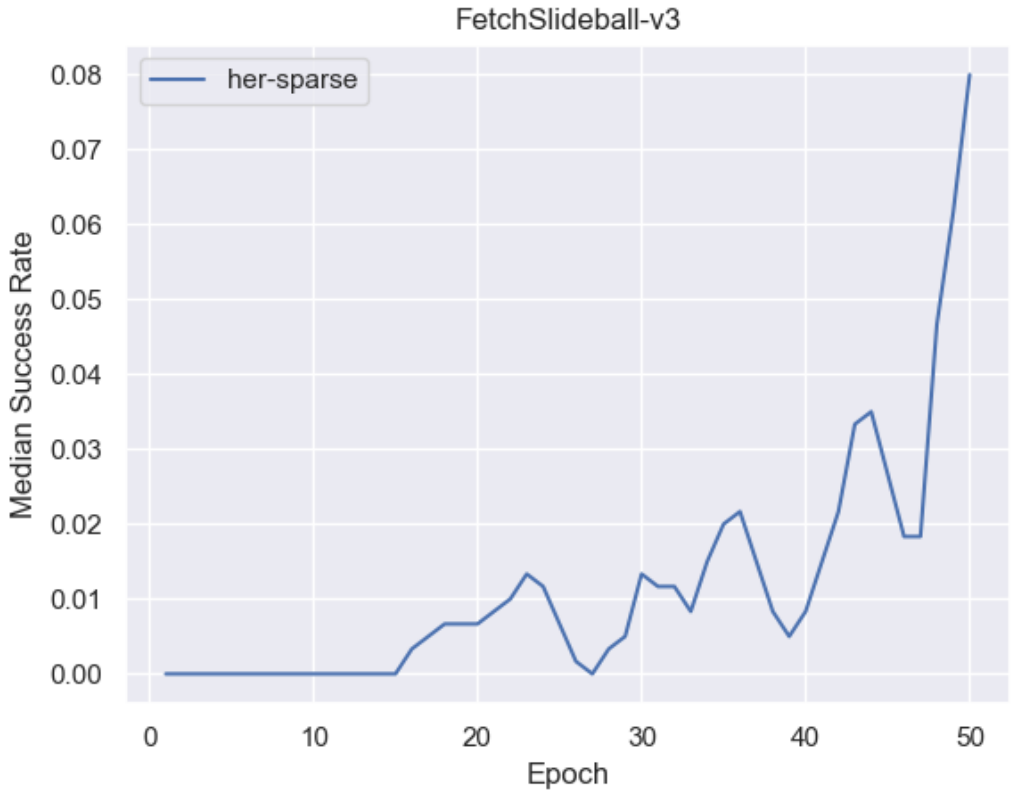


Figure 5.10.: FetchSlideball with 10% friction)

or is it only dependent on the goal being farther away ? 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 that a ball ,that touches the goal only at the end, to be counted.

5.3. 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.

5.3.1. 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.

5.3.2. Environment

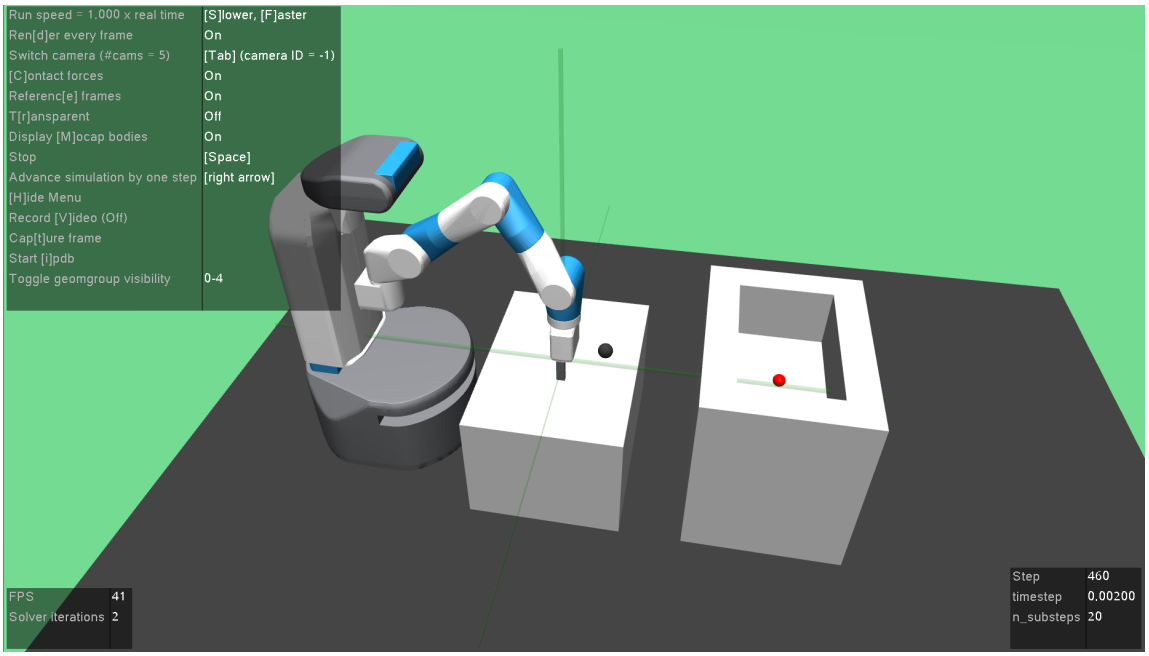


Figure 5.11.: FetchToss

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.

5.3.3. Results

As figure 5.12 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 5.13 summarizes the results for the other experiments done. The robotic somehow does not learn how to toss the ball at all. The box was changed to a box where the front is open to make it easier to toss in and a higher wall at the back 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. Also tossing a cube instead of the ball does not work. This also proved to be unsuccessful.

5.3.4. 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. Tossing a ball is pretty difficult as the results show. There can be two reasons for the agent to not show any success: Either it is just physically impossible or it is too hard to learn with Hindsight Experience Replay.

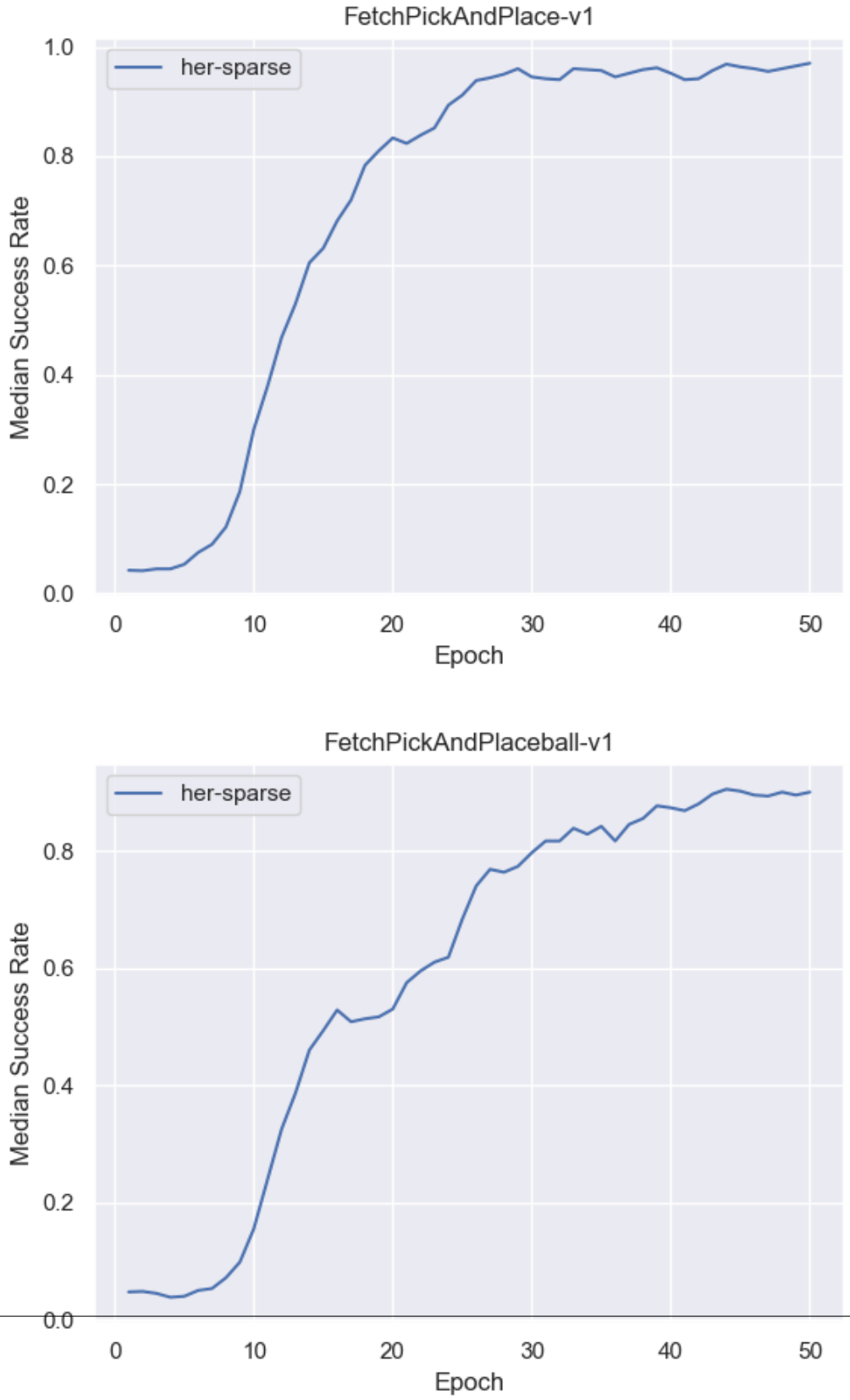


Figure 5.12.: FetchPickAndPlace with a cube(left) and a ball (right)

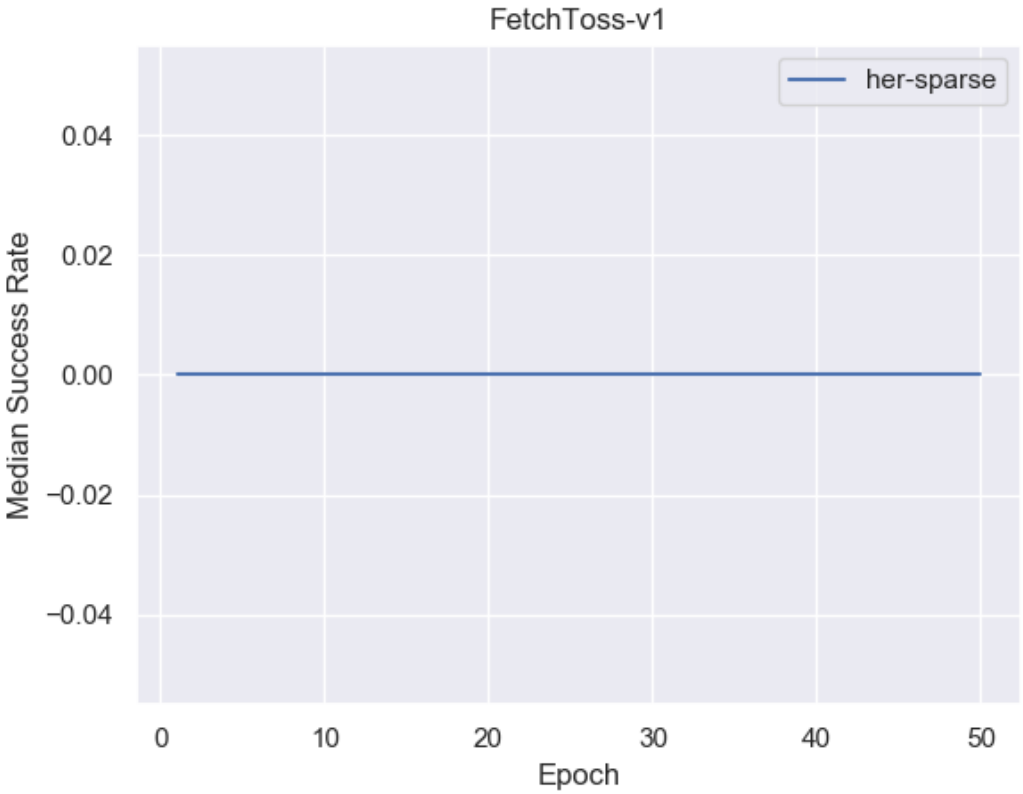


Figure 5.13.: FetchToss

6. Conclusion and Future Works

Appendix

A. Example RDMA application

```
#include <iostream>
#include <CL/opencl.h>
#include "cl_rdma.c"

using namespace std;

//transfer size 32 MB
#define N (1024 * 1024 * 32)
#define ALIGN_TO 4096

int main(int argc, char* argv[])
{
cl_uint n_platforms;
cl_context contexts[n_platforms];
cl_device_id devices[n_platforms];
cl_platform_id platformID[n_platforms];
cl_mem buffers[n_platforms];
cl_command_queue queues[n_platforms];
int altera_idx=-1;
int nvidia_idx=-1;

clGetPlatformIDs(0, NULL, &n_platforms);
if(n_platforms < 2)
{ cout<<"Found only "<<n_platforms<<" OpenCL platforms"<<endl; return(1); }

clGetPlatformIDs(n_platforms, platformID, NULL);
for(unsigned i = 0; i<n_platforms; i++)
{
      char chbuffer[1024];
      clGetPlatformInfo(platformID[i],CL_PLATFORM_NAME,1024,chbuffer,NULL);
      if(strstr(chbuffer,"NVIDIA") != NULL){nvidia_idx = i;}
      if(strstr(chbuffer,"Altera") != NULL){altera_idx = i;}

      clGetDeviceIDs(platformID[i],CL_DEVICE_TYPE_DEFAULT,1,&devices[i],NULL);
      clGetDeviceInfo(devices[i],CL_DEVICE_NAME,sizeof(chbuffer),chbuffer,NULL);
      contexts[i]=clCreateContext(0,1,&devices[i],NULL,NULL,NULL);
      queues[i]=clCreateCommandQueue(contexts[i],devices[i],0,NULL);
}
```

```
if(nvidia_idx==-1)
{cout<<"NVIDIA platform not available!"<<endl;return(1);}
if(altera_idx==-1)
{cout<<"Altera platform not available!"<<endl;return(1);}

//initialize RDMA
int rc=clrdma_init_altera(contexts[altera_idx], devices[altera_idx]);
if (rc==1){cout<<"Altera Driver is not loaded!"<<endl; return(1);}
else if(rc==2){cout<<"Wrong Altera Driver is loaded!"<<endl; return(1);}
else if(rc==4){cout<<"Could not load get_address.aocx!"<<endl; return(1);}
else if(rc){cout<<"Failed to initialize rdma for Altera."<<endl; return(1);}

rc=clrdma_init_nvidia(contexts[nvidia_idx]);
if (rc==1){cout<<"Nvidia Driver is not loaded!"<<endl; return(1);}
else if(rc){cout<<"Failed to initialize rdma for NVIDIA."<<endl;return(1);}

rc=clrdma_create_pinnable_buffer_nvidia(contexts[nvidia_idx],
                                        queues[nvidia_idx],
                                       &buffers[nvidia_idx], N);
if(rc){cout<<"Failed to create a pinnable buffer!"<<endl; return(1);}
buffers[altera_idx]
  =clCreateBuffer(contexts[altera_idx],CL_MEM_READ_WRITE,N,NULL,NULL);

unsigned long addresses[2];
rc=clrdma_get_buffer_address_altera(buffers[altera_idx],
                                    queues[altera_idx],
                                   &addresses[altera_idx]);
if(rc){cout<<"Could not get address for FPGA buffer."<<endl;return(1);}
rc=clrdma_get_buffer_address_nvidia(buffers[nvidia_idx],
                                    queues[nvidia_idx],
                                   &addresses[nvidia_idx]);
if(rc){cout<<"Could not get address for GPU buffer."<<endl;return(1);}

unsigned char* data0; unsigned char* data1;
if(posix_memalign((void**)&data0,ALIGN_TO,N))
{ cout<<"Could not allocate aligned memory!"<<endl; return(1); }
if(posix_memalign((void**)&data1,ALIGN_TO,N))
{ cout<<"Could not allocate aligned memory!"<<endl; return(1); }
for(unsigned i=0;i<N;i++){data0[i] = i%256; }
clEnqueueWriteBuffer(queues[altera_idx],buffers[altera_idx],
                     CL_TRUE,0,N,data0,0,NULL,NULL);

//RDMA transfer from FPGA to GPU of size N (timeout after 5 seconds)
rc = read_rdma(addresses[altera_idx], addresses[nvidia_idx], N, 5);
if(rc){cout<<"RDMA transfer failed. Code:"<<rc<<endl;return(1);}

clEnqueueReadBuffer (queues[1],buffers[1],CL_TRUE,0,N,data1,0,NULL,NULL);
```

```
//check whether the data is correct
for(unsigned i=0;i<N;i++)
{
      if(data0[i]!=data1[i])
      {cout<<"Transferred data is incorrect!"<<endl;break;}
}

free(data0); free(data1);
return(0);
}
```

B. Setup Instructions

This section provides an overview of the procedures needed to run the benchmarks from the source code package included for this thesis. The Ubuntu 14.04 OS is assumed.

The following commands install the Altera ICD on the system:

```
sudo mkdir /etc/OpenCL/AlteraICD
sudo cp -r altera_icd/bin/* /etc/OpenCL/AlteraICD
sudo sh -c "echo '/etc/OpenCL/AlteraICD/libalteraicd.so.1' \
    >> /etc/OpenCL/vendors/altera.icd"
```

The modified Altera PCIe module can be compiled with the following script:

```
./make_all.sh
```

The modified NVIDIA module can be compiled with the following command:

```
make module
```

To load the modified modules, the original modules have to be removed first. Since the display manager depends on the NVIDIA module it must be stopped too. To do this, the TTY has to be switched, e.g. with the key combination `CTRL+ALT+F4`. Then, the following command sequence will replace the modules:

```
sudo service lightdm stop
sudo rmmod nvidia-uvm aclpci_drv nvidia
sudo insmod nvidia.ko
sudo insmod aclpci_drv.ko
sudo service lightdm start
```

To load the modules automatically during the boot procedure, the previous modules should be overwritten:

```
sudo cp aclpci_drv.ko /lib/modules/$(uname -r)/misc/
sudo cp nvidia.ko /lib/modules/$(uname -r)/kernel/drivers/
```

For testing, the code from appendix A or the provided benchmark `direct_benchmark` can be used.

List of Figures

Bibliography

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